POLITECNICO
MILANO 1863

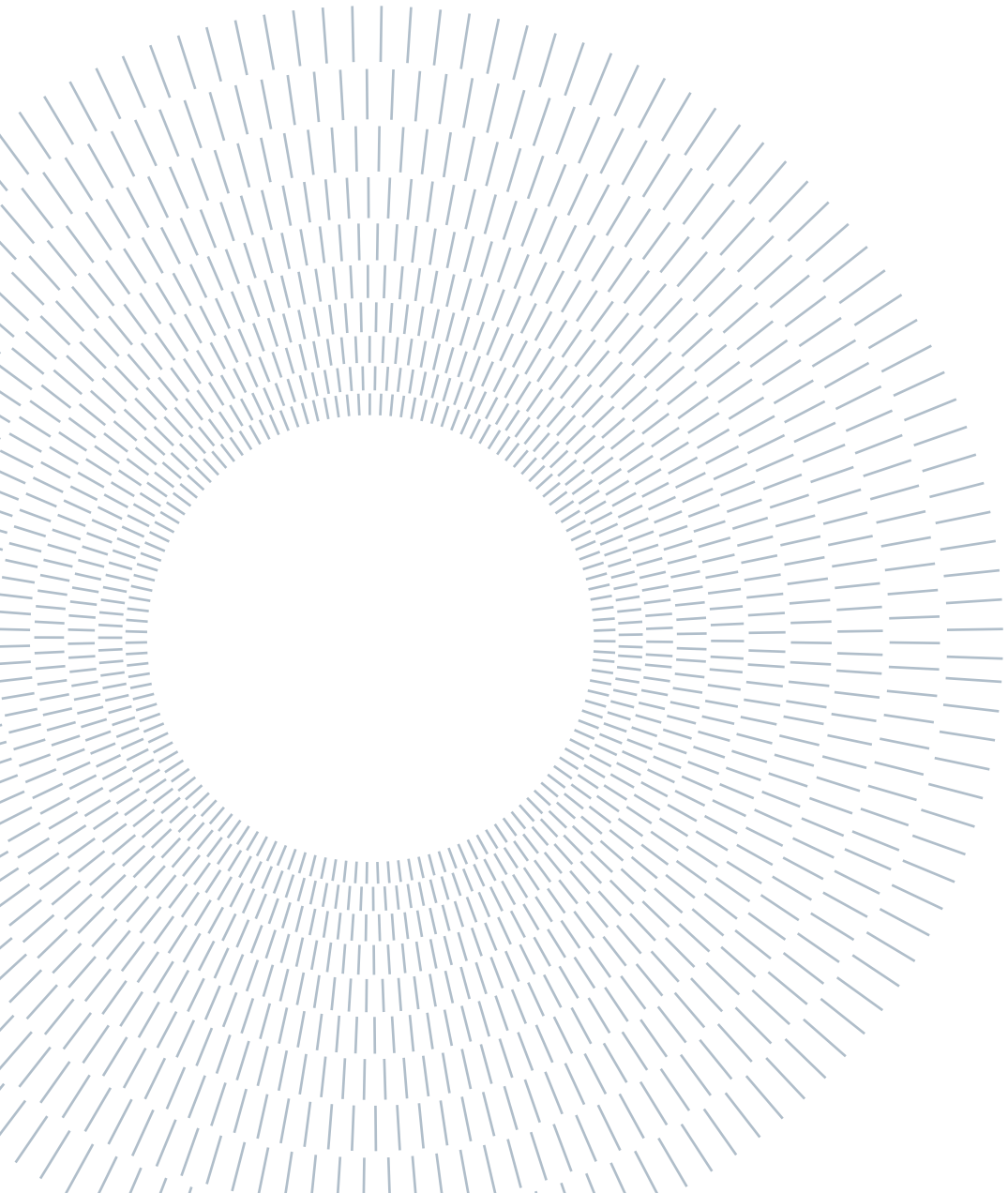
SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Analysis of Redux, MobX and BLoC and how they solve the state management problem

Tesi di Laurea Magistrale in
Computer Science And Engineering - Ingegneria Informatica

Author: **Lorenzo Ventura**

Student ID: 906003
Advisor: Prof. Luciano Baresi
Academic Year: 2021-2022



Abstract

This thesis starts defining and dividing an application state in two parts: the *ephemeral* state and the *shared* state. Then, it decomposes the state management problem in three subproblems, independently solvable. The first sub-problem is about architecting and defining an application shared state in a predictable and testable way. This is the most known problem and can be solved with patterns like Redux and BLoC. The second sub-problem is about synchronizing the shared state with the UI consistently. Stream components and observer components solve this problem efficiently. The former render the application more flexible but introduce boilerplate and the notion of stream; the latter are simpler but less flexible when the application evolves. The third sub-problem is about sharing information between components. This problem relates to the framework and to the specific context and is solved using components that propagate information automatically.
The work continues presenting patterns and approaches used to solve each state management sub-problem. Patterns and approaches are presented from a conceptual point of view before being contextualized in the Flutter framework. The objective of this part is to compose three state management solutions, one for each of the most known approaches: Redux, BLoC and MobX. Composed state management solutions are then compared in term of introduced boilerplate using two applications with different complexity. The first application is relatively small and handles a list of todos. The second application has a higher complexity and handles multiple user biometrics taken from different sources. Solutions are compared with each other and with a baseline determined using the basic features offered by Flutter to handle an application state. Before concluding, an experiment aimed at quantifying the impact of the state management solution on the application performances.

What emerges is that solutions based on Bloc and Redux introduce a substantial amount of boilerplate, whereas the solution based on MobX slightly detaches from the baseline thanks to the code generator. Another interesting fact is that the boilerplate added by all the three state management solutions with respect to the baseline is higher in the appli-

cation with a limited number of lines of code but tends to be amortized with the growth of the application in complexity and size. This conclusion, however, is not supported by a sufficiently large collection of data and should be deeper investigated. To conclude, the impact of the state management solution on the application performance ends to be negligible and constrained to be less than a threshold.

Keywords: state management, Redux , BLoC , MobX, boilerplate

Abstract in lingua italiana

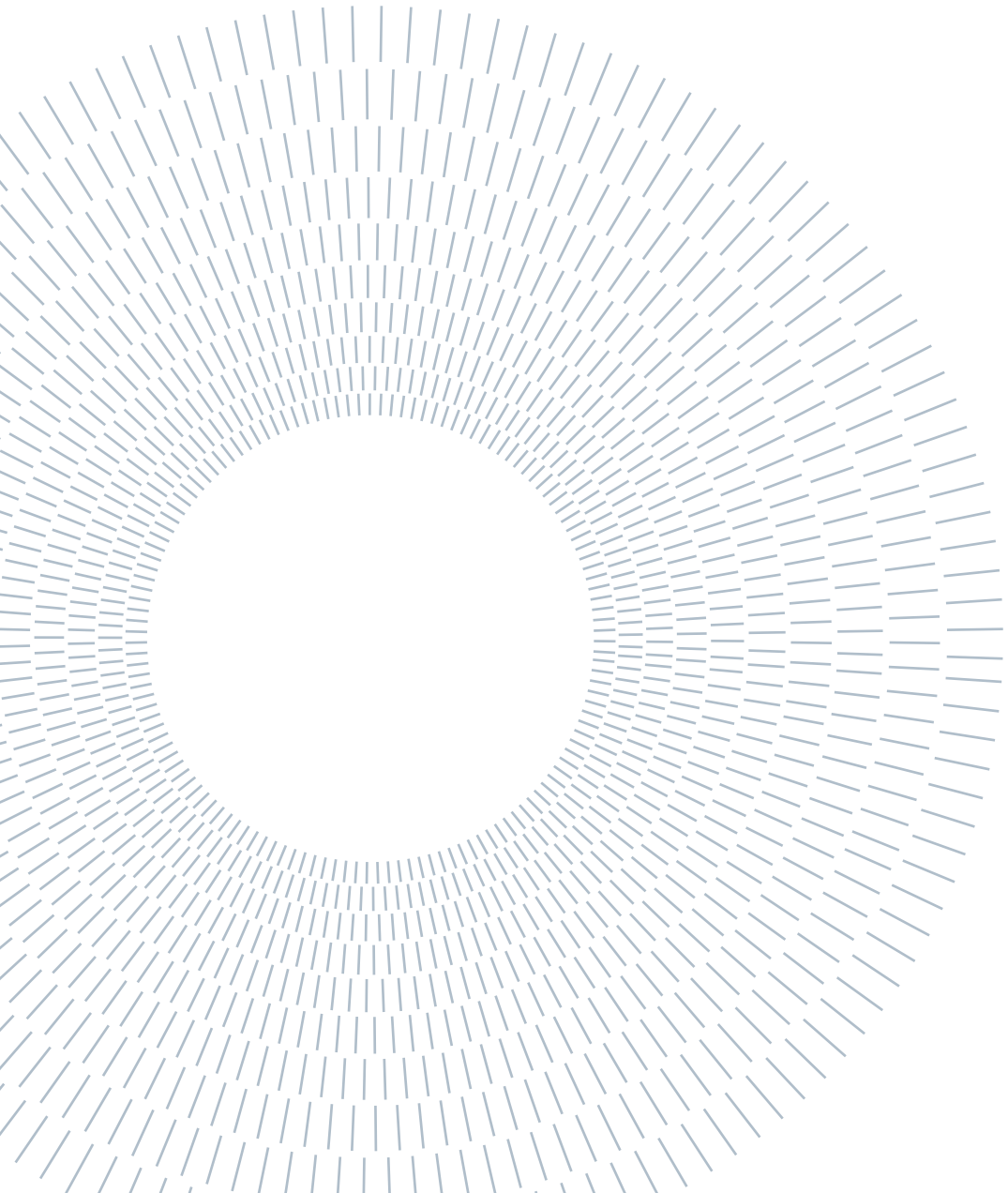
All'inizio di questa tesi si definisce e si divide lo stato di un'applicazione in due parti: lo stato *effimero* e lo stato *condiviso*. In seguito, si scompone il problema della gestione dello stato in tre sotto-problemi, risolvibili indipendentemente. Il primo sotto-problema si occupa di definire e strutturare lo stato condiviso in modo predicibile e testabile. Questo è il sotto-problema più conosciuto e può essere risolto con i pattern Redux e BLoC. Il secondo sotto-problema si occupa di sincronizzare lo stato condiviso con la UI in modo consistente. I componenti a Stream e i componenti osservabili risolvono questo problema in modo efficiente. I primi rendono l'applicazione più flessibile ma introducono anche una discreta quantità di boilerplate e la nozione di stream; i secondi sono più semplici ma meno flessibili quando l'applicazione evolve. Il terzo sotto-problema riguarda la condivisione delle informazioni tra i componenti. Questo problema si relaziona strettamente con il framework e con il contesto specifico ma, generalmente, viene risolto utilizzando dei componenti che propagano l'informazione automaticamente.
Il lavoro continua presentando una serie di pattern e approcci che risolvono ciascun sotto-problema. Questi ultimi sono presentati prima da un punto di vista concettuale e poi contestualizzati nel framework Flutter. L'obiettivo di questa parte è di comporre tre soluzioni per la gestione dello stato, una per ognuno degli approcci più conosciuti: Redux, BLoC e MobX. Le soluzioni composte sono poi confrontate in termini di boilerplate aggiunta, utilizzando due applicazioni con complessità differente. La prima è relativamente piccola e gestisce una lista di "todo". La seconda ha una complessità più elevata e gestisce molteplici biometriche relative all'utente raccolte da diverse fonti. Le soluzioni sono confrontate sia tra di loro, sia con una baseline determinata utilizzando le funzioni base che Flutter offre per la gestione dello stato di un'applicazione. Prima di concludere, il documento descrive un esperimento volto a quantificare l'impatto della soluzione per la gestione dello stato sulle performance dell'applicazione.

Quello che emerge è che sia BLoC che Redux introducono una sostanziosa quantità di boilerplate mentre, MobX si discosta leggermente dalla baseline grazie al suo generatore di codice. Un altro fatto interessante è che le tre soluzioni per la gestione dello stato

aggiungono una maggiore quantità di boilerplate rispetto alla baseline nell'applicazione con un numero limitato di linee di codice. Tuttavia, la quantità di boilerplate tende ad essere ammortizzata con la crescita della complessità e della grandezza dell'applicazione. Questa osservazione, tuttavia, non è supportata da una quantità di dati abbastanza elevata e andrebbe investigata più a fondo. Concludendo, l'impatto della soluzione per la gestione dello stato sulle performance dell'applicazione risulta trascurabile e limitato a una soglia massima.

Parole chiave: gestione dello stato, Redux , BLoC , MobX, boilerplate

Contents



1 | Introduction

1.1. Purpose

Applications complexity significantly increased during the last 10-20 years. This growth highlighted the importance of having an effective tool to handle application states and their visualization. Common design patterns, like MVC, were no longer enough to embrace all the dynamics aspects of complex mobile applications, this caused new techniques, called state management solutions, to come out in great number.

State management solutions adopt different approaches and provide several benefits in multiple aspects of the development process of complex mobile applications, however, they all tend to introduce the so called *boilerplate*.
In computer programming, *boilerplate code* or *boilerplate* refers to sections of code that have to be included in many places with little or no alteration. [17]

This thesis builds the foundations for a comparison between state management solutions but also for the definition of a common model that collects/detects their common aspects. The target point of view for the comparison is the boilerplate they introduce but other aspects, such as the impact of a state management solution on the application performances, are also transversely touched.

1.2. Organization

This document is organized in 7 chapters (beside this one). Starting chapters are devoted at introducing the state management problem and the state management solutions considered in the comparison. Subsequent three chapters describe the implementation of three applications used to test different aspects of the state management solutions.

- Chapter 2 introduces some important concepts. In particualr, it defines what is:
  - an application state,

  - a component state,
  - the difference between stateful and stateless components,
  - a state management pattern,
  - a state management library,
  - a state management solution,
  - the separation of concerns and its benefits.

  Chapter 2 also introduces the Flutter framework and the state management problem. About the latter, it arguments over why it is so important before dividing it into three subproblems that can be solved independently. Chapter 2 also explains the two most common patterns used to limit the drawbacks of the raising applications complexity: immutability and unidirectional data flow.

- Chapter 3 introduces several state management patterns and libraries that will be used, in the subsequent chapters, to carry on the comparison. This introduction is purely conceptual, it targets the approach that stands behind the solution without considering any particular implementation. In the last part, libraries and patterns are collocated in a Venn diagram based on the subproblem they solve.

- Chapter 4 gives a practical perspective of the patterns and libraries introduced in Chapter 3. It considers their implementation in the Flutter framework highlighting their pros, their cons and their most important concepts using UML diagrams and two simple examples. The end of the chapter is devoted at composing three state management solutions with the patterns and libraries previously defined.

- In Chapter 5 starts the data collection process about the boilerplate. It explains in detail the implementation of an application that handles a list of todos using the state management solutions composed in chapter 4. The end of each section reports a table showing the collected data.

- Chapter 6 presents the implementation of another, more complex, mobile application using the state management solutions composed in chapter 4. The application has a reasonably large shared state and deals with a mocked up external device. The end of the chapter reports a table regarding the lines of code used by each state management solution.

- Chapter 7 proposes an experiment that quantifies the impact of the synchronization process (of an externalized state) on the application performances. The applica-

tion performs a sort of "stress test" of the synchronization process to stipulate its maximum additional cost.

- The conclusive chapter merges the collected data about the boilerplate of the three implementation processes using a histogram and formulates some conclusions. The end of the chapter suggests the possible future development of this work.

NOTE: I will use a tool called CLOC [1] throughout the entire thesis in order to count the lines of code produced by different the applications.

The tool can be used running the following command in a terminal

cloc (application-directory/lib)

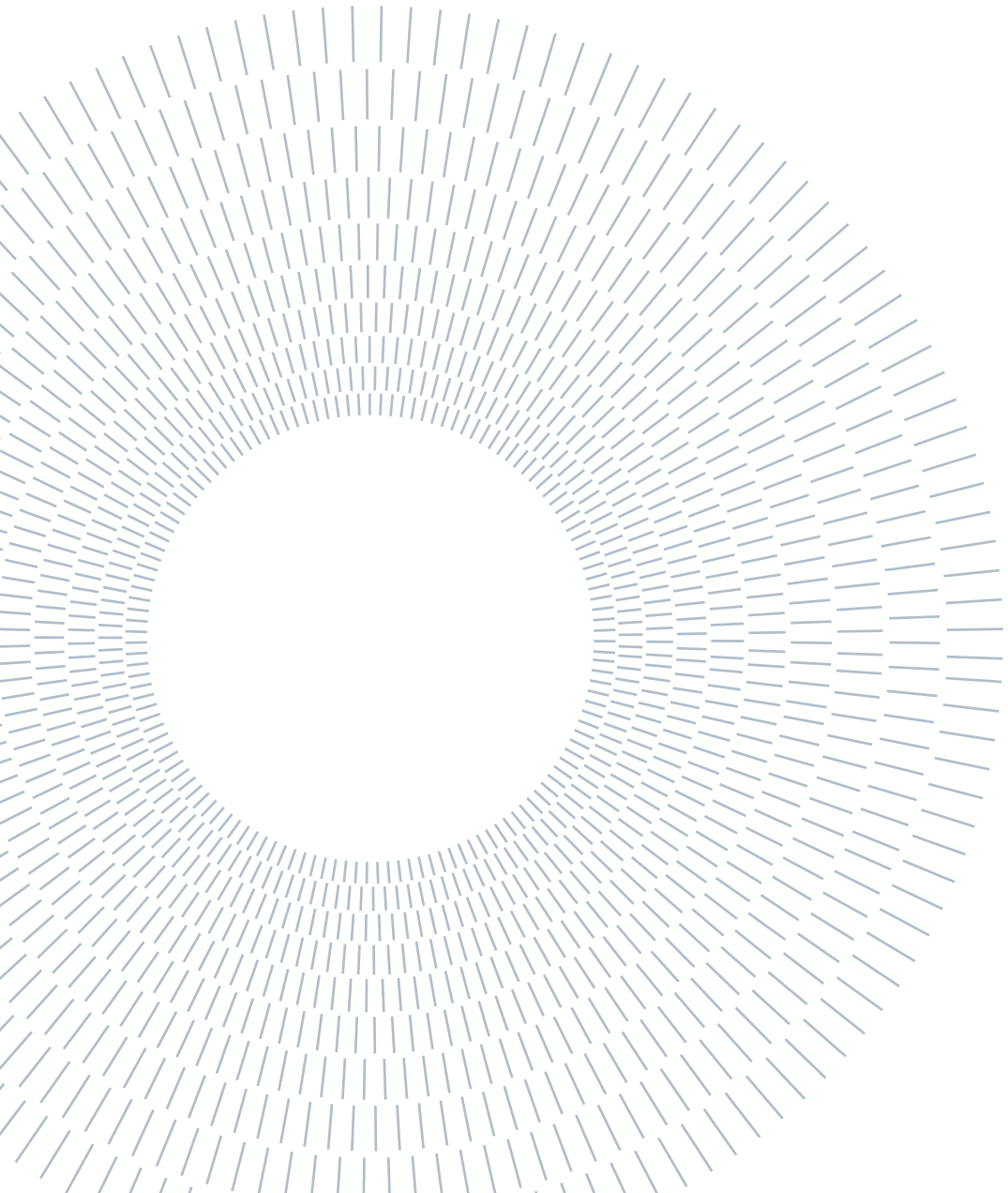
and produces a summary as the one below (for example):

31 text files.
31 unique files.
0 files ignored.

github.com/AlDanial/cloc v 1.92 T=0.19 s (164.8 files/s, 9252.7 lines/s)

Language	files	blank	comment	code
Dart	31	197	5	1538
SUM:	31	197	5	1538

The column indicated as "code" reports the lines of code omitting the comments and black lines and is the one I use to carry on the comparison.



2 | The state management problem

This chapter introduces the basic concepts to carry on the comparison between state management solutions throughout the thesis. It starts introducing Flutter; its characteristics, how it structures user interfaces and the problems it tryies to solve. Then, it defines the state management problem and list some useful definitions.

2.1. Flutter framework

On the surface, Flutter is a **reactive, pseudo-declarative UI frameworks,** in which the developer provides a mapping from an application state to its interface, and the framework takes care of updating the interface at runtime when the application state changes.

In most traditional UI frameworks, the user interface's initial state is described once and then separately updated by user code at runtime, in response to events. One challenge of this approach is that, as the application grows in complexity, the developer needs to be aware of how state changes cascade throughout the entire UI. For example, consider the following UI:

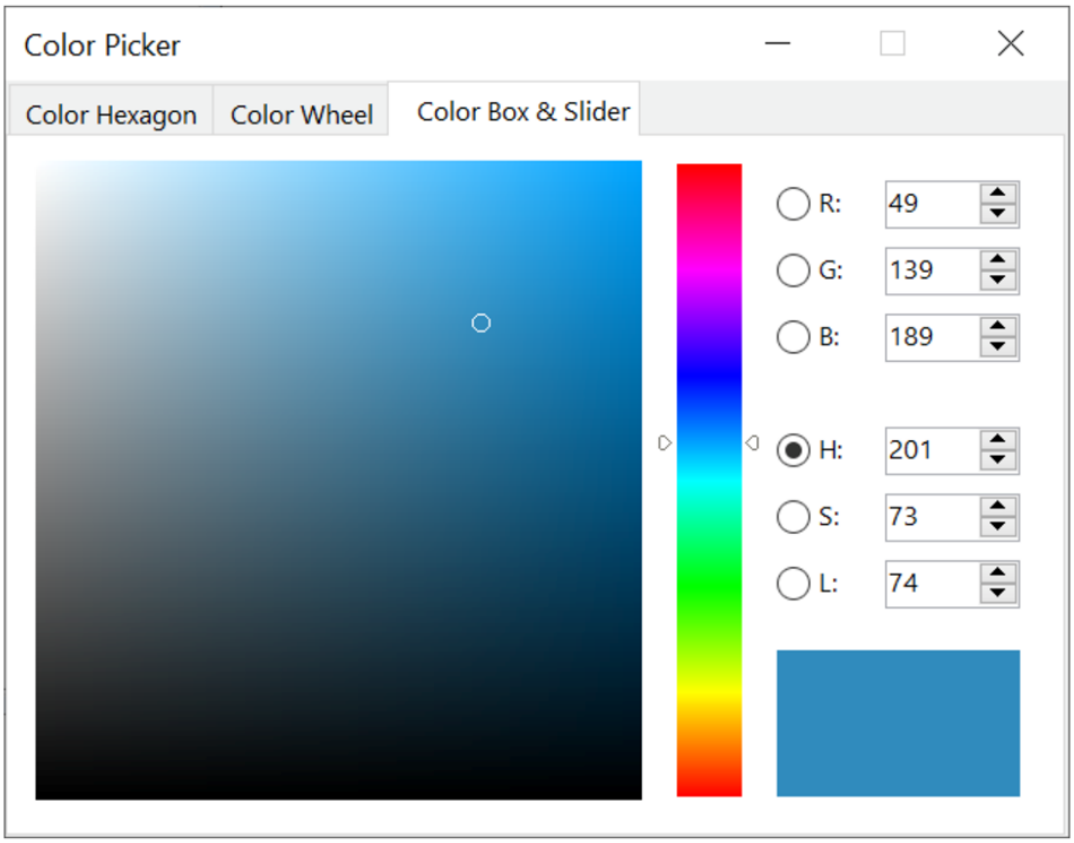


Figure 2.1: Example of complex UI [11]

There are many places where the state can be changed: the color box, the hue slider, the radio buttons. As the user interacts with the UI, changes must be reflected in every other place. Worse, unless care is taken, a minor change to one part of the user interface can cause ripple effects to seemingly unrelated pieces of code.

One solution to this is an approach like MVC, where you push data changes to the model via the controller, and then the model pushes the new state to the view via the controller. However, this also is problematic, since creating and updating UI elements are two separate steps that can easily get out of sync.

Flutter, along with other reactive frameworks, take an alternative approach to this problem, by explicitly decoupling the user interface from its underlying state. You only create the UI description, and the framework takes care of using that one configuration to both create and/or update the user interface as appropriate. [11]

2.2. Components

Components is the general term given to the base blocks used to build user interfaces in the context of mobile applications. Components can be of various types: layer components, visual components etc. Components are used to visualize part of an application state.

Flutter framework layers components as tree structure. This really helps at managing complexity.

2.3. State

A state, in the broadest possible sense, is **a representation of a system in a given time.**

The **state of an application** comprehends everything that exists in memory when the app is running. In practical terms, it is all we need to rebuild it and its behaviour in any moment in time. This includes the app's assets, all the variables, animation state, textures, fonts, network calls state, database calls state, timers, and so on. However, some states are not managed by you (like textures). The framework handles them. So, a more practical definition would be: "an application state is everything that exists in memory when the app is running **that you manage**." [12]

Components are used to visualize parts of an application state. If we want our application to behave dynamically, we need to keep track of the state of components.

The **state of a component** is whatever data is needed to rebuild it and its behaviour in any moment in time. An application state is composed of one or more component states (plus much else).

For example, imagine an application that displays a list of filtered todos based on a filter (completed, not completed or both). It contains a component that displays the filtered list from which one can be selected. The application state is composed of:

- The entire list of todos
- The filter
- The filtered list

- The index of the current selected item
- Lots of other stuff not handled by you

Whereas the state of the component is only made up of the actual filtered list and the index.

A component state refers to the information needed to the specific component to work properly. An application state is a wider concept, it usually contains information that is never visualized.

An application state can be separated into two conceptual types: ephemeral state and shared state.

- **Ephemeral state** (sometimes called UI state or local state) is the state that can neatly fit in a single component. Other componets seldom need to access this kind of state. There is no need to serialize it, and it doesn't change in complex ways. In our previous example, an ephemeral state could be the current selected item. It is necessary to the component to work properly but is not required by other components.
- **Shared state** is a state that is not ephemeral, which is shared across many parts of the app, and that is kept between user sessions. In our previous example, a shared state could be the list of todos. Other examples of potential shared state:
  - User preferences
  - Login info
  - Notifications in a social networking app
  - The shopping cart in an e-commerce app
  - Read/unread state of articles in a news app

There is no clear-cut rule, you can decide that all the state of your app is ephemeral. It goes the other way, too. For example, you might decide that, in the context of your particular app, the selected tab in a bottom navigation bar is not ephemeral state. You might need to change it from outside the class, keep it between sessions, and so on. In that case, the _index variable is a shared state.

There is no an universal rule to distinguish whether a particular variable is ephemeral or app state. Sometimes, you'll have to refactor one into another. For example, you'll

start with some clearly ephemeral state, but as your application grows in features, it might need to be moved to shared state [12].

For that reason, take the following diagram with a large grain of salt:

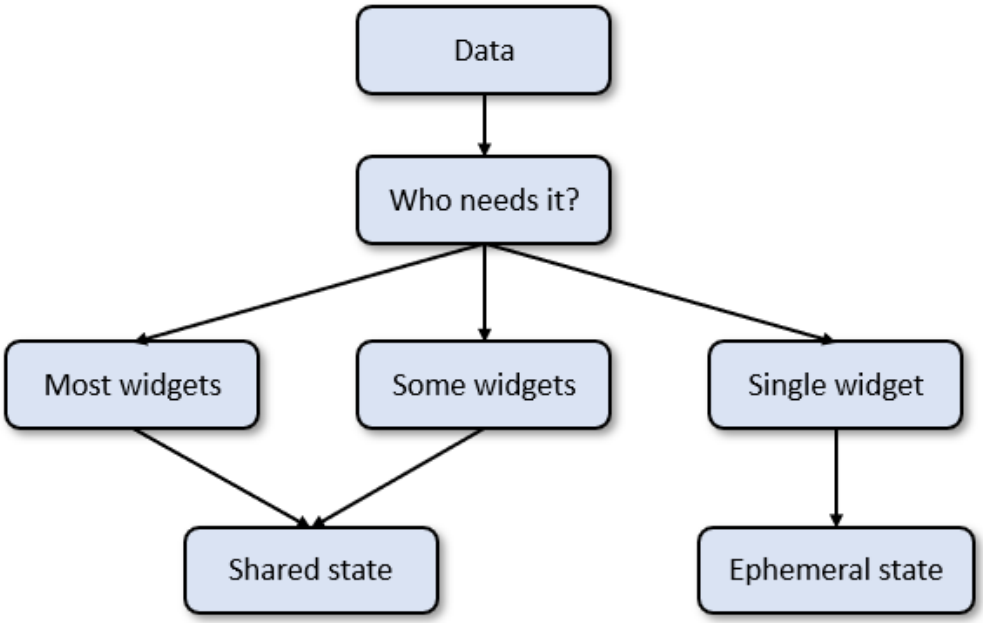


Figure 2.2: Decisional diagram for ephemeral and shared state [12]

2.4. Stateful and Stateless components

Components can be divided into two categories: stateless and stateful. The first ones are not associated with any state, they just receive immutable data to be visualized and are used to represent the parts of the state that never change over time (static). The latter are associated with a mutable state and are used to represent the parts of the state that change over time. If we want a component to behave dynamically, we need to use a stateful component.

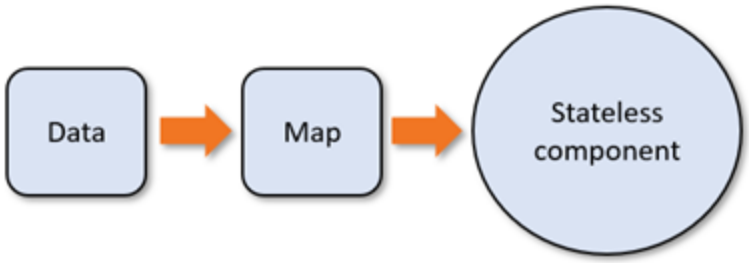


Figure 2.3: Data flow to create a stateless component

Stateless components are created with some input data (or just none) and remain static. They cannot be mutated without being destroyed and rebuilt.

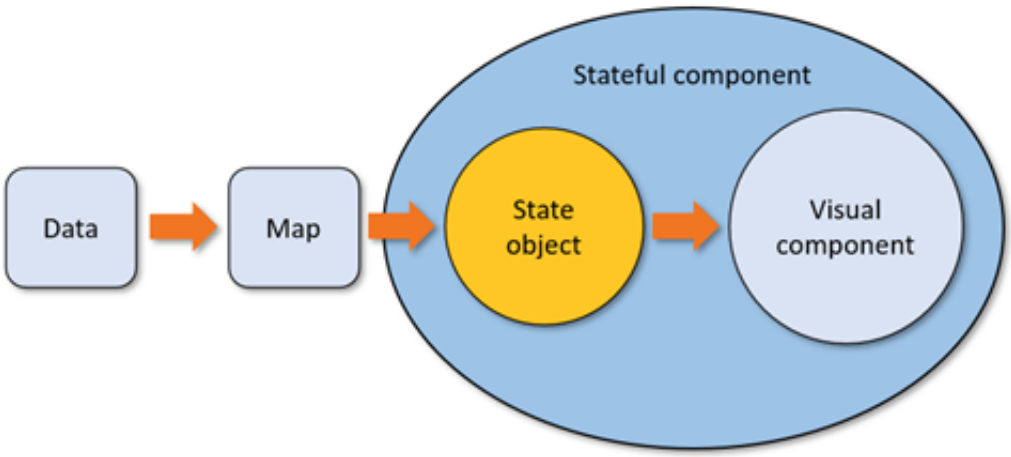


Figure 2.4: Data flow to create a stateful component

Stateful components are used to expose the mutable parts of the application state to the user. A stateful component is composed of a state object, which represents the internal mutable state, and a visual component, which displays it. A state object is frequently mutated during an application lifecycle. Its corresponding visual component is kept in sync by the framework, which monitors state objects, multiple times per second, efficiently. In case of a state object change, it updates the corresponding visual component.

2.5. An application state example

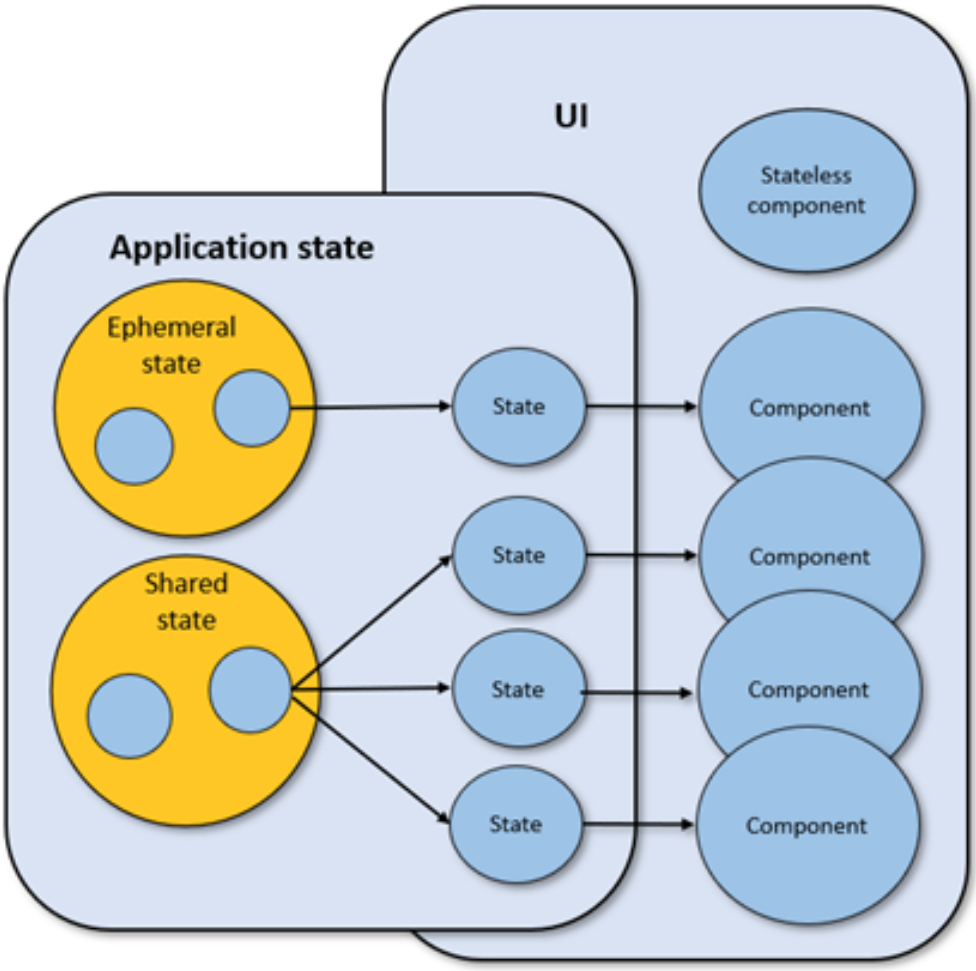


Figure 2.5: Example of an application state at runtime

Try visualizing the entire scenario with an image and an example. Figure 2.5 refers to a specific moment in an application lifecycle. UI shows five components, four are stateful and one is stateless. Application state contains four component states, three of which deals with a part of the shared state, the remaining one with a part of the ephemeral state. Other parts of the shared and ephemeral state are not currently visualized (by the UI).

This image could, for example, represent the state of a todo application. The four stateful components are:

- One for the list of filtered todos
- Another for the current filter and its mutations
- Another for the application theme
- Another for tab switching

The first three handle a shared state, the last one an ephemeral state. As we said, there isn't a general rule to distinguish which part of the state if ephemeral or shared. In this case, diagram 2.2 is taken as reference. For example, the application theme is shared because is used by more than one component. The current tab state, whereas, is used by a single component, for example the homepage, and can neatly fit inside a single state object. When the tab changes, it is the only one rebuilding, whereas, when the theme changes, all components in the page rebuild.

The stateless component could represent the application title. It is static and does not change.

The parts of the shared state not being visualized could be the status of the authentication or the plain list of todos. They are supposed to be used by more than one component and kept between sessions but are not associated with any state object momentarily. An example of the not visualized ephemeral state could be the status of a checkbox in the previous page. It is inherent to the correct functioning of checkbox only. It is kept in the application state to be used later, when the current page is popped.

2.6. What state management is and why we need it

State management is a technique of structuring, dispatching and synchronizing an application state throughout components of the application.

When an application state grows, it also tends to get messier. A state stored in a single place grants more control but prevents to be dispatched to multiple components efficiently. On the other hand, splitting or duplicating it requires a lot of effort to keep it synchronized with the rest of the application and with itself. In general, the complexity of an application grows exponentially with its state. Therefore, state management solutions mostly target complex applications instead of small ones.

To scale an application without losing control on its complexity is necessary to define

one or more standard approaches to interact with state, its mutations, and its synchronization. These approaches can vary from design patterns, tools, packages/libraries, or just guidelines.

Two well established principles that help to handle a complex state are: **Immutability** and **Unidirectional data flow**. They are sort of guidelines and introduce some costs. In the long run, however, their cost is amortized by the benefits they introduce in terms of predictability. Flutter frequently use them to keep things consistent and clean. So do packages like Redux and BLoC. Other packages (MobX) decided to follow a lighter approach with a mutable state.

The first principle is **immutability**, which means that we should never mutate data directly without creating a new reference of that object. If we mutate data directly, our application becomes unpredictable and it's really hard to trace bugs. An immutable object is an object whose state cannot be modified after it has been created. If you want to modify some properties of an object you have to do it on a copy of the object.

The second principle is **Unidirectional data flow**. It is also known as one-way data flow, which means data has only one way to be transferred to other parts of the application. In a nutshell, it is the absolute owner of that specific piece of state that oversees updating it (immutable of course). In essence, this means child components are not able to update data coming from the parent component and neither send data to it. The whole system gets a lot of benefits from using Unidirectional data flow; however, it also limits information transfer between components. [3]

Overall, the state management problem introduces three questions:

- **How is state?** (1)
  How is it composed, shaped, architected? How are its internal mechanisms defined? Where is state located? How can be mutated? Structuring an application state in an efficient and predictable way is really important. It helps with testing, bug avoiding and team working. In general, it boosts the implementation and maintenance process.
- **How state is propagated down the tree?** (2)
  How is state accessed by components? How components access other components' state? Layering UI components as trees has become a standard. It has multiple advantages but presents an issue. Components can only propagate information

downward to children due to unidirectional data flow. Imagine two components on the same level, but distant from each other, that need to access the same state. There is no possibility for them to communicate besides lifting state to the nearest common ancestor. This practice is commonly called "lifting state up". Let's visualize it with an example.

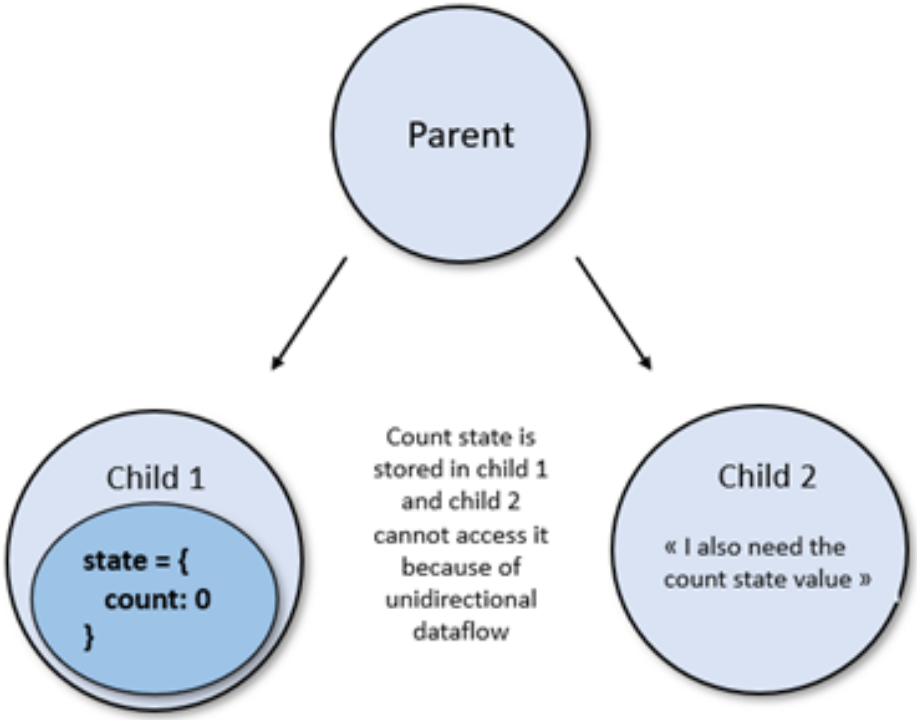


Figure 2.6: Example of "lifting state up" [7]

Component one contains a counter state. Suddenly component two needs to access the counter value. However, component one and component two cannot directly communicate. Counter state must be lifted to the parent component and passed downward to both children. This practice, mixed with unidirectional data flow, gives birth, in big UI trees, to a phenomenon called "Props Drilling". It refers to the necessity to set up a lot of connections between components to make information flow down the tree. Imagine that the nearest common ancestor, between component one and component two, is hundreds of levels above. Lifting state up requires creating at least *2 x numberOfLevels* connections!

- **How are UI components kept in sync with state? (3)**
  This question is more practical with respect to the previous ones. Building UIs

that stay coherent with their underlining application state is fundamental. This synchronization usually requires so much effort that current technologies are trying to render it automatic. This limits errors but, on the other hand, does not provide lot of flexibility. The way in which components are kept in sync with state really depends on the environment we are working on and on the framework itself. State management patterns, usually, do not answer this question themselves or just partially, whereas packages do. For example, some patterns say:

" information is sent to UI in form of an object through a stream"

but does not specify how that stream is used to keep UI updated.
This question is usually solved by the implementing package. Packages use specific features offered by the framework to link state with UI in a sort of automatic way.

2.7. State management solutions

State management packages/libraries and **state management patterns** are two different concepts.

A state management pattern is an abstract concept, it provides **a conceptual solution** to one or more state management questions.

A state management package/library provides **a practical solution** to one or more state management questions **in a given environment.**

For example, store an application state in a unique place and mutate it through predefined actions is a state management pattern. It helps dealing with state and its mutations. A state management package could implement this pattern providing useful functions to define its mechanism in the Dart language (for example).

State management packages often implement a specific state management pattern, but this is not always the case.

A state management solution uses one or more state management packages and/or patterns to solve **all** the state management questions.

Comparing state management patterns requires a higher level of abstraction with re-

spect to packages/libraries. The latter are more suitable for quantitative analyses. They can be measured in performance, boilerplate, and many other aspects.

2.8. Separation of concerns

The concept of SOC helps with managing complexity in large applications. In computer science, **separation of concerns (SoC)** is a design principle for separating a computer program into distinct sections. Each section addresses a separate *concern*, a set of information that affects the code of a computer program.

Why is it so important to divide code in sections? Here some advantages:

- **Faster development process:** SoC supports rapid and parallel development. If Soc is used, a group of developers can work on the view while another can work on the business logic.
- **Interchangeable view and logic:** With SoC you can create multiple views for a model but also different models for a view.
- **High Testability:** because components belong to specific layers in the architecture, other layers can be mocked or stubbed, making this pattern relatively easy to test
- **Lack of duplication and singularity of purpose:** SoC provides a sort of divide and conquer approach; each layer addresses a part of the entire domain. Since complexity grows exponentially, separating domain in smaller pieces really breaks down the overall complexity. On the other hands, breaking down small domains leads to an increment in complexity.

Layered architecture also has some disadvantages, but they are amortized by the growth of the application. In general, can be said that:

"the usage of SoC helps to control and encapsulate the complexity of large applications, but adds complexity to small ones."

3 | A conceptual perspective

I propose, throughout this chapter, a series of approaches each targeting a specific question of the state management problem and, for each of them, I list a series of advantages and disadvantages. In the final part, I propose a way of combining various state management patterns and libraries to obtain complete state management solutions.

3.1. setState

This approach is used to synchronize the view of a stateful components with its internal state. It is based on modifying state objects through a specific method offered by the framework, called *setState*. The *setState* method takes as payload a state changing function, called *callback* function. Once a state object is modified with *setState*, the framework is able to recognize the change rebuilding the corresponding component.

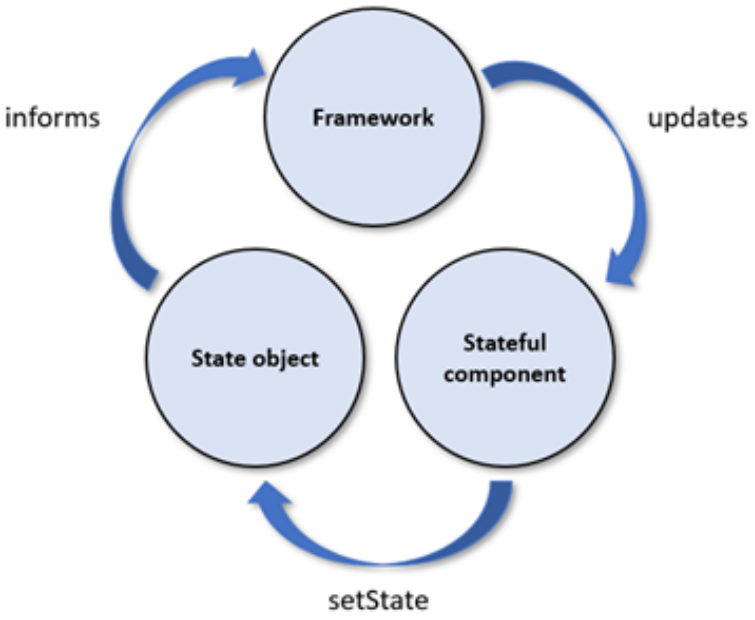


Figure 3.1: *setState* flowchart

Figure 3.1 shows how changes are propagated using setState. Notice the unidirectional data flow. Actors always forward information to the next element and never to the previous one.

The setState mechanism is both simple and powerful at the same time. It is used, under the hood, by most of the libraries and high-level components. It provides a naïve but effective reactive programming mechanism that automatically propagates state changes to the UI.

"setState" answers the question: How is UI kept in sync with state?

In fact, to mutate a state using setState method is enough to keep it synchronized with its visualization.

SetState process is asynchronous. An arbitrary time can pass from when the setState method is called and when the UI is updated. Even if Flutter framework ensures that state objects are mutated synchronously, the propagation of changes to the visualization is asynchronous. Calling *setState* method basically sends a request to the framework to apply changes contained in the callback function. Since setState is just a request, the framework can decide to process it immediately or to postpone its execution. This comes with a lot of benefits in terms of performance and memory optimization. Indeed, Flutter is built with the capacity to sustain an elevated number of setState calls and component renderings per second.
However, *setState* asynchronous behaviour can introduce very subtle bugs if not handled properly.

"setState" method presents a big issue; it leads all the dynamic components in the subtree to rebuild. This behaviour coupled with the practice of lifting state up has a huge impact on performance and memory optimization in complex applications. Here's an example:
Imagine having two dynamic components that share a part of the state. Let's say, given a mutable number, one shows it doubled and the other shows it halved. The two components are on the same tree level and, therefore, require the state to be lifted to their common ancestor; for simplicity, their parent. However, their parent already has its own state, for example a checkbox, not related with the number. Figure 3.2 shows the (simplified) component tree related to this scenario.

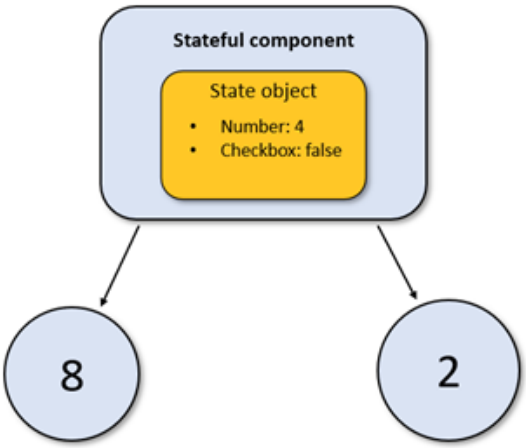


Figure 3.2: A stateful component containing a shared state

Imagine changing the common number (using *setState(number++;)*). Both the parent and the children rebuild. This occurs even if the parent component was not showing the common number at all. This is not ideal; three components rebuild when only two should. An even worse scenario occurs when the checkbox value changes. All three components rebuild even if only one was affected by the change.
This was a simple example just to expose the problem, but imagine this behaviour in a context where children are two huge, nested components made up of thousands of levels. This creates an enormous waste in memory and performance. To clarify, it is important to state that the framework does not blindly rebuild components without checking if they changed. The correct way of saying this is: the *build* method of each component in the sub-tree is called every time a state change occurs in the state object.

setState features:

- No external libraries / completely handled by the framework (+)
- Easy to use (+)
- Reactive
- Asynchronous
- Rebuilds all the sub-tree

3.2. Storing application state in state objects

State objects are efficiently handled by frameworks and are supposed to contain state. Why can't we just insert our application state into state objects?

Theoretically, we could, but as the application state grows multiple issues are introduced:

- **Props drilling problem**
  Props drilling problem is introduced in Section 2.6. State information should flow smoothly from one component to another to make an application scalable and the implementation process linear. Since state objects are layered as a tree, forwarding props from one state object to another can become really tiresome in big tree structures.
- **No soc** (section 2.8)
  State objects are linked with stateful components and the framework itself. Storing an application state in state objects makes it dependent on its visualization. For example, testing an application which stores state in state objects requires instantiating one or multiple components just to test the logic.
- **Different lifecycles and handled by the framework**
  State objects are created and destroyed multiple times during the application lifecycle, whereas application state can remain unchanged. State objects are handled by the framework and its internal mechanism. Application state usually deals with different types of procedures, like network calls, database calls, timers etc. These procedures can be blocking, time consuming and asynchronous. State objects, on the contrary, should remain as light as possible due to their dynamic behaviour.

In my opinion, these issues arise because two separate concepts are combined. State objects are created with the objective **to contain the state of a stateful component.** (A component state refers to the **information necessary to the specific component be visualized and work properly**). An application state is a wider concept, it relates to information of various types inherent with the application domain. Sometimes this two concepts overlap, sometimes don't.

In some cases, parts of the application state are suited to be contained in state objects; like the ephemeral state, it does not change in complex ways, and neither is accessed by other components. Conversely, other parts, like shared state, require a more optimized and ad hoc mechanism to behave as intended. They are not suited to be contained in

state objects because they would introduce the issues presented above.

Storing an application state (or part of it) in a separate container solves the issues listed above by introducing all the benefits related with soc. Some of them are really useful in the context of large applications, like predictability. However, it also introduces some drawbacks:

- **Synchronization with state objects**
  The framework requires information to reside in state objects to be visualized and behave dynamically. Even if state is moved elsewhere, sooner or later it must be injected into state objects. These state object must be updated when the application state changes. Overall, this new state container needs to be synchronized with state objects, which already need to be synchronized with components, adding complexity as shown in figure 3.3.

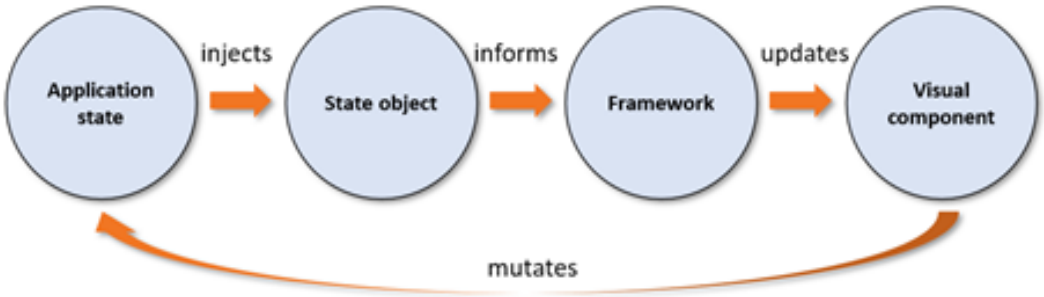


Figure 3.3: Synchronization of an external application state with its visual component

- **Increase complexity and boilerplate**
  This drawback relates closely to the previous one. Separating state from state objects inevitably increases the number of lines of code to handle the whole mechanism.

3.3. Plain setState plus state objects

Handling an application state with setState and state objects presents different conceptual issues:

- **Low scalability**
  Due to unidirectional data flow, information can travel only downward the tree; Moreover, the effort spent to make information reach the destination is proportional

to the travelled levels/distance (the more levels the more connections must be wired). Also, for unidirectional data flow, providing a state to two different components requires it to be lifted to their nearest common ancestor. In general, the more a state is shared throughout components the more it needs to be lifted up.
On the other hand, growing in complexity, an application also grows its UI tree and its shared state. Due to this phenomenon, the shared parts tend to move upward to reach a higher number of components, whereas their deepest destination tends to move downward (see figure 3.4).

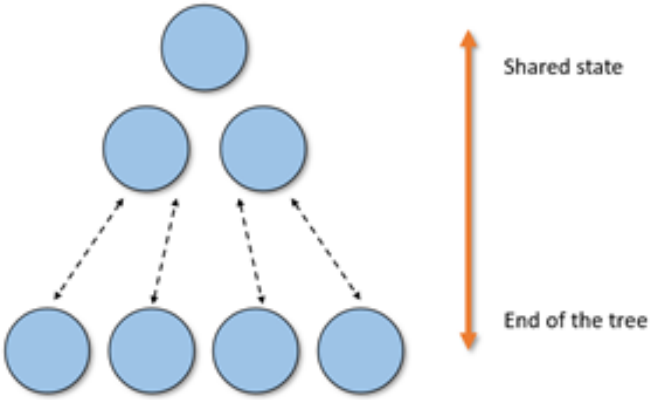


Figure 3.4: Component tree evolution with complexity

Overall, the mix of the raising distance to be travelled and the expense of moving information downward makes the use of plain setState with state objects unfeasible in big scenarios in terms of scalability. A minimal change or feature addition requires a huge effort.

- **Low Performances (Flutter)**
  To make matters worse, setState does not provide a mechanism to reduce wasted component renderings. Every time a state object is mutated, all components in the sub-tree rebuild. As an application grows in complexity, state tends to move upward in the tree increasing the number of components rebuilt at any state change. Memory consumption and application performances will soon get out of hand.

The listed issues arise from these concrete problems:

- Lack of a method to dispatch state efficiently
- Impossibility to perform partial rebuilding on the component tree (mostly in Flutter)
- Melting of the business logic layer with the presentation layer

- State objects are too simple to handle the shared state

3.4. Using context (Providers and InheritedWidgets)

A way to solve the "props drilling problem" is to use context to propagate state, or in general data, down the tree.
Flutter propagates data down the tree, by default, for its internal mechanisms. This information is encapsulated in the so-called context, or context objects. Context objects can be used to propagate state down the tree instead of wiring up every single connection from parents to children. If information is inserted in these context objects correctly, it is made available to all components in the sub-tree, automatically.

Usually, special components are offered by the framework to do the job for us.

A component providing a value to its sub-tree using context is called *provider*. Components accessing the state of a provider through context objects are called *dependents*. Figure 3.5 shows an example of a value dispatched through a provider and two dependents accessing it from the sub-tree.

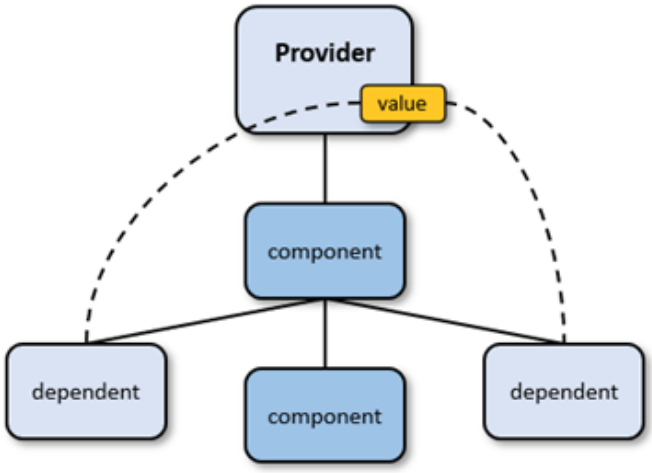


Figure 3.5: Example of a shared state provided to the sub-tree using context [5]

Contrary to the other state management approaches that provide trade-offs to aspects of the state management problem, propagating state with context works effectively, and is a well-established solution. It is simple and free of any tangible drawback. The only issue I see with this approach is that information is exposed to the whole sub-tree and not to

the interested parts only. Manually wiring connections allows a fine-grained information exposure. However, even if this can be a warning from a security point of view, it is not a big deal in the context of mobile application development.

3.5. Observer components

SetState alone does not solve the problem of keeping UI in sync with state exhaustively. It requires state to reside in state objects which are, on the other hand, not suited to contain complex states. If a state is moved elsewhere, it requires, sooner or later, to be injected in one or more state object to be visualized by the framework, introducing another layer of synchronization. It is vital to render this binding as automatic and consistent as possible. The ideal scenario would be to have a state container, shaped and architected at our own discretion, in pure Dart, and to be able to interact with it without caring about its visualization. This would ease the implementation process solving one of the largest sources of bugs. Component diagram 3.6 shows an observer component connecting an observable source of data to a stateful component.

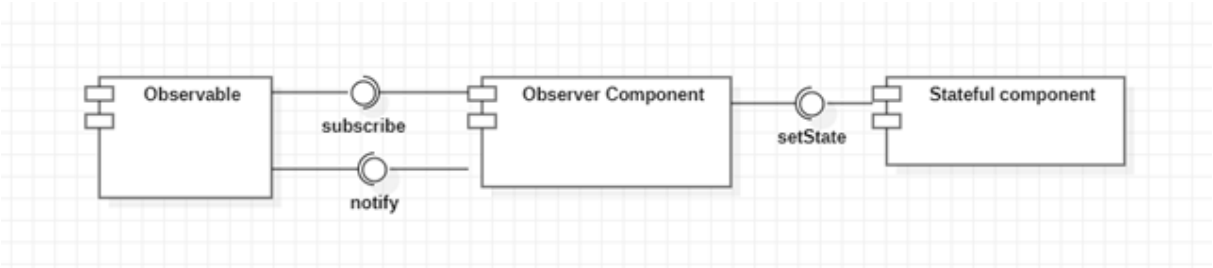


Figure 3.6: Observer component component diagram

An observer component is a component which can directly relate with **an external source of data** (or in our case state). An observer component subscribes to an observable entity and gets notified (and re-rendered) when the observable changes. Of course, more than one observer component can subscribe to the same observable source. The observable source must store a list of dependents to be notified upon a change and the observer component must store a reference to the data source.

3.6. Stream components

Stream components are a way to convert elements of a stream into their visual representation, automatically. A stream component is associated with a stream. It is provided with a mapping that relates elements coming out from the stream to a visual representation.

Once a new element appears in the stream, the stream component re-renders automatically. Component diagram 3.7 shows a stream component connecting the output of a stream of data to a stateful component.

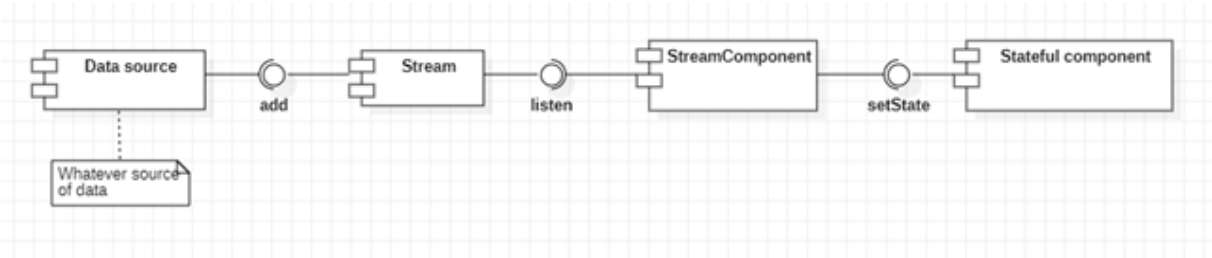


Figure 3.7: Stream component component diagram

They are similar to observer components because they both provide a way to listen and react to changes of an external data source. They can be used to help the synchronization process bind an external source of data (the state) to its visualization representation (visual components) automatically.

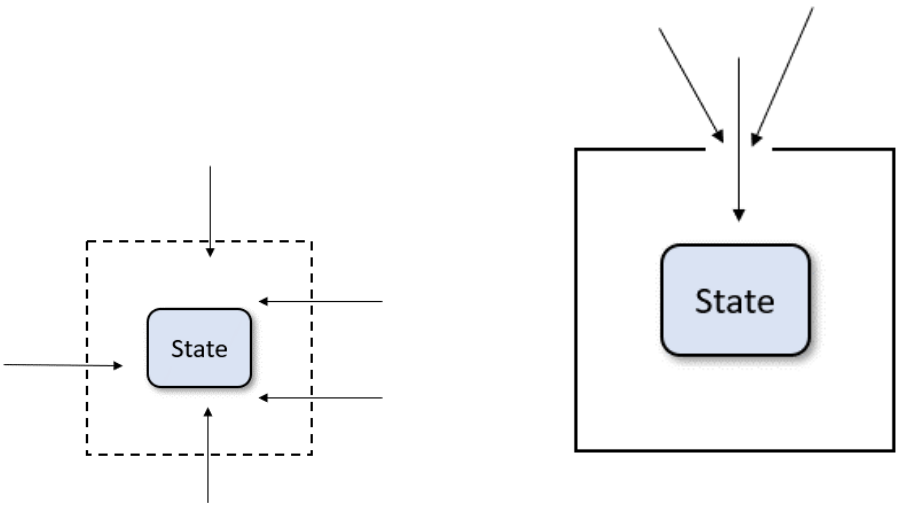
Stream components are really powerful because they are independent from their data source. While an observer component must subscribe to a specific data source, a stream component subscribes to data coming out of a pipe, independently of the data input from the other side/end.

Imagine an observer component that listens to a data source, and that you decide to substitute the data source with another one. This implies directly changing the observer component definition and adapting it (make it subscribe) to the new source of data. Stream components work differently. You can substitute their source of data without updating the stream component at all. They do not store any information about the data source, they only store a reference of the output stream independently who is inserting data. Moreover, the data source does not need to store a list of dependents to be notified on a change, it just pushes new data into the pipe.

Stream components **boost code reusability and flexibility**. With stream components, the business logic layer can be substituted easily without modifying the visualization layer. However, relying on stream, they are generally **harder to use** and generates a lot **more boilerplate** with respect to observer components.

3.7. Action-based mutations

This approach consists of mutating state only through predefined, allowed mutations. This is a really powerful approach and a "must have" in the context of complex applications. Try visualizing it with the example in Figure 3.8.



(a) Direct mutations (b) Action-based mutations

Figure 3.8: Mutating an application state

Figure 3.8a represents a state not using actions. We can think of it like a house whose walls are substituted with pillars. Figure 3.8b represents a state using actions. Think of this like our usual concept of house. It has a unique entrance, and is surrounded by walls that forbid passage. Imagine monitoring people who enter the house in order to filter and count them or just to check on who enters the house. In the left scenario, it is way more probable that you are going to miss someone in the count because people enter the house from multiple places at the same time/simultaneously. In the right scenario, the probability that you miss someone is basically zero. You can just stand at the door waiting for people to enter, one after the other, and count them.

The same behaviour applies to managing a state in the context of complex applications. If state is mutated frequently and directly in more than one aspect, it is really probable that some mutations pass unseen and generate unexpected behaviour. If mutations come sequentially through a specific entrance (a stream for example) and are encapsulated into

predefined containers (object, string etc) specifying what kind of change will be introduced into the state, the whole scenario becomes clearer and more predictable. We can log mutations and register a copy of the state before and after the change.

This architectural design mixed with immutability is powerful because it allows every state transaction to be analysed and the cause of an unexpected behaviour to be understood. Once a crash or bug occurs, we can determine, simply by looking at the state transaction, if the problem was generated from an inconsistency in the business logic. If we observe a correct transaction, the issue was for sure generated somewhere else, probably in the visualization layer.

3.8. Redux

Redux was created by Dan Abramov and Andrew Clark in 2015. Redux objective is to create a predictable container for the application state. It is a design pattern that operates in a similar fashion to a reducing function, a functional programming concept.

Redux focuses on solving the question: How is state?

It has three core principles [14]:

- **Single source of truth**
  The state of the application is stored in an object tree within a single store. It does not require the entire application state to be stored in the store, however, if it doesn't, it loses all the benefits of Redux.
- **State is read-only**
  The Redux store adopts action-based mutation patterns introduced by section 3.7. The only way to change the state is to emit an action, an object describing what happened. Actions never mutate state, they produce new ones with the help of reducers. (immutability) This ensures that neither the view nor network callbacks will ever write directly to the state. Instead, they express an intent to transform the state. Because all changes are centralized and happen one by one in a strict order, there are no subtle race conditions to watch out for. As actions are just plain Dart objects, they can be logged, serialized, stored, and later replayed for debugging or testing purposes.
  Usually, a unique doorway is provided to input actions. The Store provides a public method to be called with an action as payload.

- **Changes are made with pure functions**
  Redux boosts predictability introducing pure state mutations. This means that an action must mutate state in a deterministic way without generating any side effect. Reducers are pure functions that specify how the state tree is transformed by actions.

A reducer takes the previous state and an action and returns the next state. Reducers can be split into smaller reducers that manage specific parts of the state tree. Reducers apply changes synchronously. Figure 3.9 shows the architecture of a Redux store.

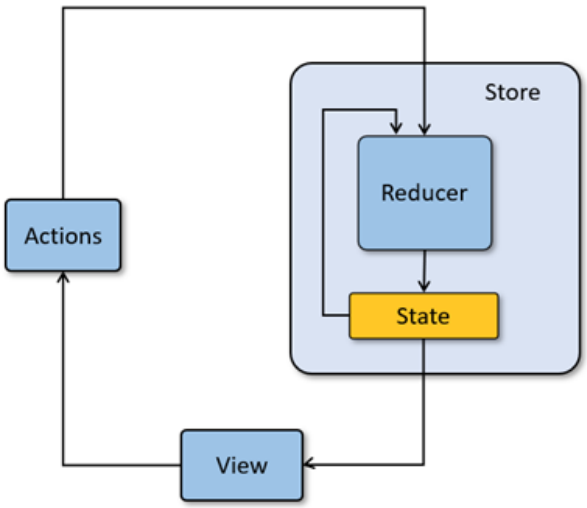


Figure 3.9: Architecture of a system using the Redux pattern [13]

For example, image handling a counter value with the Redux pattern. The counter value is contained in the store in the current state. Two actions are defined: IncrementAction and DecrementAction. The current state holds the value 0. The workflow is the following:

- The view emits a new IncrementAction in the store.
- The reducer takes the current state (0) and the IncrementAction and generates the new state (1).
- The view updates

The whole process is completely deterministic. However, due to this pure way of handling state mutations, Redux requires external/additional tools to handle asynchronous operations. The most common choice is to use Middlewares. Middlewares are just functions and act as a sort of proxy processing the action before it reaches the reducer. One or

more middlewares can be stacked up to be executed before actually calling the reducer. Middlewares can have multiple tasks; the most common one is to handle asynchronous operations. An async operation is split into one or more actions (usually at least two). Let's visualize the process of fetching a file from a web API.

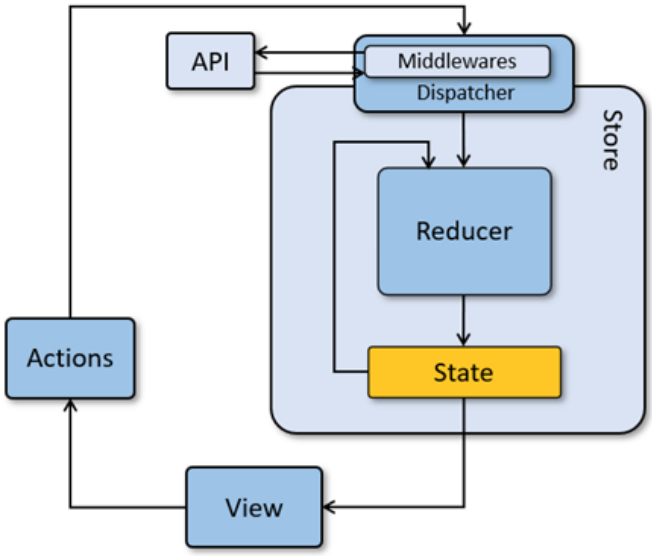


Figure 3.10: Architecture of a system using the Redux pattern with middlewares [13]

Figure 3.10 shows the Redux architecture upgraded with the mechanism of middlewares. Sequence diagram 3.11 describes the procedure of fetching a file from a web API using the architecture proposed in figure 3.10.

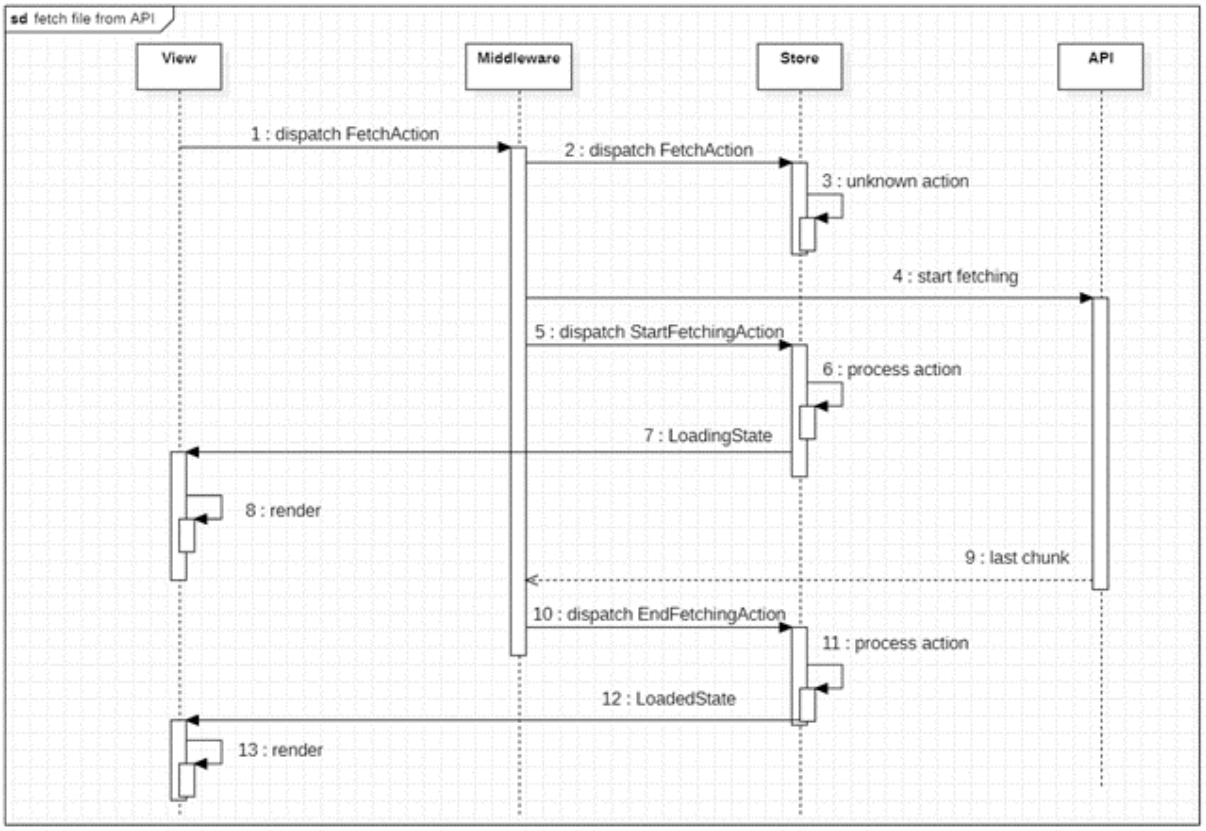


Figure 3.11: Fetching a file from a web API

1. The intention of fetching data from a service comes into the middleware as an action object: *FetchAction*
2. The middleware propagates the action to the next middleware (in our case there is only one middleware so the action is sent directly to the reducer)
3. The reducer receives the *FetchAction* which, however, does not correspond to any of the possible actions it can handle. Consequently, it discards the action and keeps listening for new ones
4. The middleware starts the fetching process.
5. Right after the API call it emits another action called *StartedFetchingAction*
6. The Store receives the *StartFetchingAction* which, this time, it handles by producing a new loading state
7. The Store sends an event of type *LoadingState* to the view
8. The view layer is informed of the state change and re-render consequently
9. The middleware receives the last file chunk

10. The middleware emits a new event, called *EndFetchingAction*, with the file as payload
11. The store processes the action producing a new *LoadedState*
12. The Store send the *LoadedState* to the view
13. Visualization triggers again and re-renders ending the process

Note: the process is not completely sequential. The Store and the Middleware execute in parallel. For example, step 3 and step 4 can occur simultaneously. The same is for step 6 and 9.

A consideration concerning centralized state - Centralized state introduces multiple features which can be useful or harmful depending on the context [18]:

- **No encapsulation** - When a value is inserted into the store, it is visible to every component that can access the store.
- **Accessible and dispatchable fields** - all necessary information resides in a single place.
- **As state grows normalization is needed** - accessing deep nested data frequently is expensive. Normalization flattens data and makes it more accessible in performance but requires effort to be performed.
- **Storing state and its history is easy** - Having all information in a single object (or almost) allows snapshots of the state to be easily created.
- **Component state persists even after it has been dismounted** - Let's say we have a ShoppingCart component, and that we need to share some data about it, for example, isCartFull. So we put isCartFull in the global store. Now let's say that we fill up our shopping cart with products and continue to the purchase page. Eventually we pay, and the shopping cart component is no longer needed. What happens now is that the global store still holds a variable isCartFull that is now set to true, and we must remember to clear this flag once a new ShoppingCart component enters the screen. We must manually keep cleaning the store from garbage of old components that have already left the screen. This can lead to countless bugs.

NOTE: Redux only cares about creating a predictable container for the application state. The way in which this container is used and bound to the UI is left to the specific implementing package/library.

Redux features:

- Functional
- Pure
- Centralized state
- Predictable (+)
- Separation of concerns (+)
- Design pattern
- Plain Dart (+)
- Explicit (+)
- Immutable
- Synchronous, need adjunctive tools to handle asynchronous actions (-)
- Hard to learn (-)
- Potentially lot of boilerplate (-)

3.9. Bloc

The BLoC design pattern was designed by Paolo Soares and Cong Hui, from Google and first presented during the DartConf 2018 (January 23-24, 2018).

Bloc answers the question: How is state?

BLoC is an acronym for Business Logic of Components. It aims to move the business logic to one or several BLoC s and remove it from the Presentation Layer. The BLoC pattern was initially conceived to allow reusing the very same code independently of the platform: web application, mobile application, back-end.

Streams are the foundation of Bloc. It relies on exclusive use of Streams for both input (Sink) and output (stream) of the business logic layers. These constraints allow a business logic to remain independent from the platform and from the environment.

In short, when a component sends something to a Stream, it no longer needs to know [2]:

- what is going to happen next,

- who might use this information (no one, one or several Widgets...)
- where this information might be used (nowhere, same screen, another one, several ones...),
- when this information might be used (almost directly, after several seconds, never...).

Overall, Bloc also attempts to make state changes predictable by regulating when a state change can occur, using events, and enforcing a single way of changing the state throughout an entire application.

A bloc is an object that receives events and emits states. Figure 3.12 shows the architecture of a logic layer composed of a single bloc.

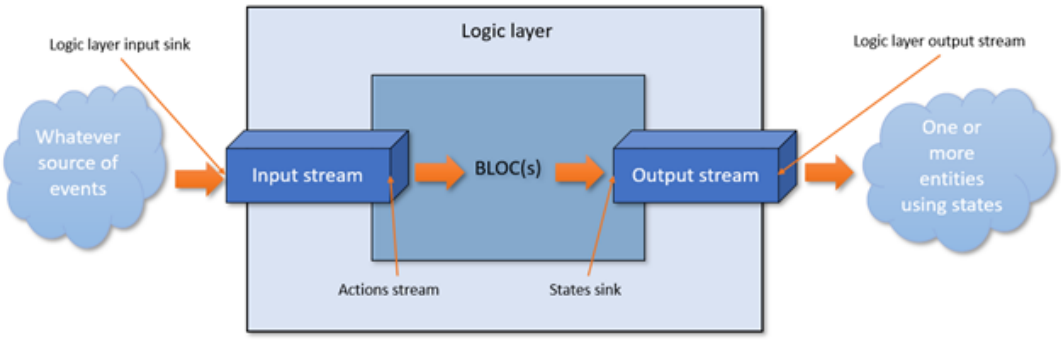


Figure 3.12: Architecture of a logic layer using the BLoC pattern

States and actions are immutable Dart objects. A bloc receives events through an input stream and consumes them to emit new states into an output stream. Depending on the implementation, the current state can be stored to be accessed later. A bloc can have dependencies on one or more blocs to react to their state changes. Contrarily to Redux, blocs can operate on actions impurely. They can perform async operation, side effect etc.

The usage of the BLoC pattern allows us to separate our application into three layers:

- **Presentation** (UI)
- **Business logic** (blocs)
- **Data**

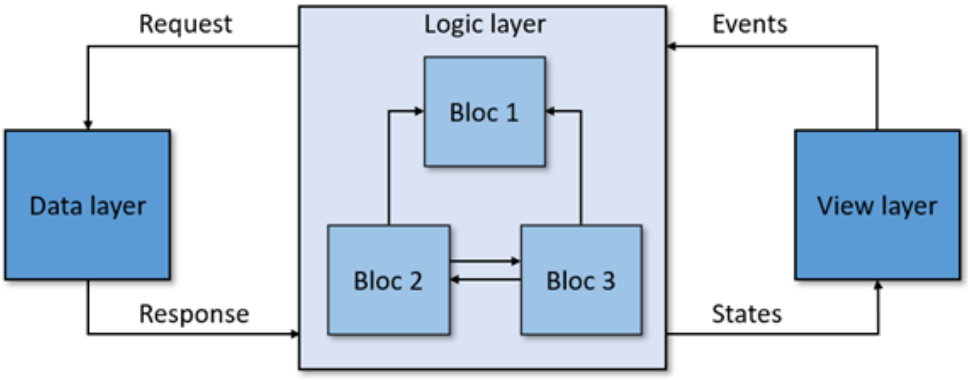


Figure 3.13: Three layers architecture using the BLoC pattern

Figure 3.13 shows how to divide an application architecture into three layers. The view communicates with the logic layer using events and states. The logic layer retrieves information from an arbitrary data source, potentially asynchronous. Blocs inside the logic layer communicates with streams. Arrows represent blocs dependencies. For example, Bloc 2 depends on Bloc 3 and 1. Bloc 1 is independent.

Conceptually Bloc does not present any particular drawback. However, we can list these potential issues:

- the concept of streams can be quite hard to manage. The whole application becomes strongly asynchronous. Dealing with asynchrony is usually tricky for humans.
- The usage of streams for both input and output suggests that the code may be rich of boilerplate

Bloc features:

- Separation of concerns (+)
- Strong stream usage
- State modularity(blocs)
- Reusability (+)
- Design pattern

- Hard to learn (-)
- Plain Dart
- Potentially lots of boilerplate (-)
- Predefined way to change state (events)
- Immutable

3.10. MobX

Mobx was created by Michel Weststrate and is a state management library. I will try to extrapolate its core concepts without going into implementations details.

MobX answers the questions: How is state? How is UI kept in sync with state?

MobX adopts a Transparent Functional Reactive Programming (TFRP) approach implemented using the observer pattern.

MobX starts with the idea that a state can be divided into two parts: **core-state** and **derived-state**.

Core state refers to the state inherent to the business domain being processed. Derived state is computed using core state.

The classical example considers a Contact entity composed by a name, a surname and a full name. Name and surname represent the core state, whereas full name is composed using the name and surname and represents the derived state.

It is essential to keep the core state as lean as possible because it is the part that is expected to stay stable and grow slowly during the lifetime of the application.

Core state is mutable. Derived state depends on the core state and is kept up-to-date transparently by the MobX reactivity system.

First, let's define the core concepts of MobX [15] :

- **Observable state**. The observable state is determined by the combination of the core state and a derived state.

- **Actions**. Actions are the primary means to modify state. Actions can only mutate core state.
  It is important to say that MobX does not force the usage of actions. It provides the action mutating mechanism and suggests using it, but nothing prevents developers from avoiding using it.
- **Computed values**. Any value that can be computed using a function that purely operates on other observable values. They can depend not only on the core state but also on other derived states.
- **Reactions**. A reaction is similar to a computed value, but instead of producing a new value it produces a side effect. Reactions bridge reactive and imperative programming for things like printing to the console, making network requests etc...

Computed values and reactions are both referred to as **derivations**.

MobX is *"doubly reactive"*. It keeps derived state in sync with core state but keeps UI in sync with observable state too. The whole process is done transparently. But what does transparent term really mean? asked this question, Michel Weststrate answered:

"Where in normal FRP you have to explicitly subscribe to observables, and need operators to combine the different subscriptions, this is done implicitly in the case of MobX and other TFRP libraries; while running your reactive functions it observes which data you access, subscribes to them and, in that way, constructs a graph of nodes that depend on each other." [16]

He also states in another article when speaking about the MobX approach:

"Working with subscriptions (or cursors, lenses, selectors, connectors, etc) has a fundamental problem: as your app evolves, you will make mistakes in managing those subscriptions and either oversubscribe (continue subscribing to a value or store that is no longer used in a component) or undersubscribe (forgetting to listen for updates leading to subtle staleness bugs). In other words; when using manual subscriptions, your app will eventually be inconsistent. ... A minimal, consistent set of subscriptions can only be achieved if subscriptions are determined at run-time." [15]

About observer and stream components, we stated that the former are easier to implement/use because they do not require setting up a stream. However, they connect the source of data with the utilizer in a binding way because the observable must keep

a reference to every dependant. Moreover, dependants must subscribe and unsubscribe to the observable, and this is almost always done manually and at compile time. Michel Weststrate, with his transparent reactive approach, aims to automatize subscriptions to observables and to derive them at run- time.

All this has great advantages [15]:

- it becomes simply impossible to ever observe stale derivations. (No more inconsistency in what is shown). It is no more possible to oversubscribe and undersubscribe.
- Memory optimization (possible only if dependencies are determined at runtime). The minimum number of subscriptions is kept in any moment.
- Less effort for the developer
- Less coupling between the observer and the observable. One could work on the observer and the observable independently and even substitute them without paying attention to wiring up connections. The whole binding is done implicitly by the package. We can say that automatic observer components (MobX provides) act like stream components but keeping the observer component benefits.

Overall, MobX library brings the state management problem to a higher level of abstraction. It provides a well-structured black-box approach that solves the problem of keeping state in sync with itself and the UI. Developers only need to focus on defining state, its logic and its visualization. Basically, MobX takes care of the entire state management problem (almost). This is acceptable by the fact that it is more a library than just an approach. However, we need to remember that pre-packaged solutions usually present these issues:

- There is no control of what happens inside the black box – It is not clear precisely when data is changed and how the mechanism works without studying its implementations details
- They are not flexible (if the solution does not fit some particular case there is nothing you can do)
- Your project may become even more dependent on external entities (a great part of the project is dependent on the library)

MobX features:

- Is a library

- Separation of concerns (+)
- Mutable state/ impure
- Implicit update logic (-)
- Synchronous
- Easy to use (+)
- Reactive
- Transparent
- Based on the observer pattern

3.11. Putting all together

The state management problem concerns keeping state coherent and synchronized with the UI but also addresses other, more practical, issues related to how frameworks are built.

We listed some approaches, each handling one or more aspects of the state management problem.

The first questioned approach, setState, defines a way of keeping components coherent with their internal state in a reactive programming fashion. It is very effective but requires the state to be contained in state objects, which are not flexible and predictable enough to accommodate an application state (mostly the shared parts). If we move the state to a separate location (to take advantage of the benefits of SOC principally) we need special components that inject state into state object and do all the setState calls. These components can be of various types, and we have highlighted two very common ones: the observer components and the Stream components. They both have advantages and disadvantages but provide an essential mechanism to keep the business logic layer in sync with the view.

We then analysed two architectural patterns for defining an application state: Redux and BLoC. Redux approach emphasizes predictability. It conveys all the information into a single place and mutates them purely based on a set of predefined actions. BLoC emphasizes code modularity/reusability architecting the application state as a collection of blocs that communicates, both internally and externally, with streams. Redux and BLoC are two strong architectural patterns that, however, tend to generate some unwanted boil-

erplate.

Here comes MobX that takes the problem onto a larger scale. It provides a transparent state synchronization both with itself and with the view, almost completely boilerplate free. Under the hood, it uses the Observer patter.

Finally, we addressed the props drilling problem related to how Flutter architects its component tree. The solution is to propagate information downward using context and works pretty well.

Figure 3.14 arranges approaches on a Venn diagram composed of three sets. Each set represents a specific problem to be solved in the state management context.

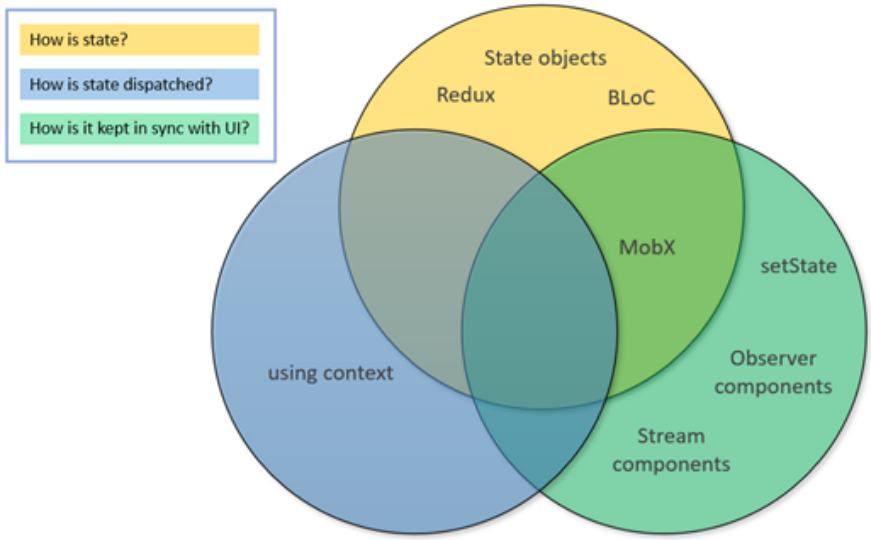


Figure 3.14: State management sub-solutions on a Venn diagram

Approaches locate based on the problem they target. Intersections contain approaches that target more than one sub-problem. We can build, using this diagram, state management solutions made up of various approaches. To compose a state management solution, it is necessary to choose at least one approach for each set. Some examples are:

- **Solution 1**: using context + state objects + setState
- **Solution 2**: using context + Mobx
- **Solution 3**: using context + MobX + Redux

- **Solution 4**: using context + BLoC + Stream components

NOTE: placing MobX was hard. Initially I intended to place it in the green set (sync set) because it mostly targets the problem of synchronizing the state. However, it also targets some aspect of the state definition. It provides annotations to wrap state into observables and to define action-based mutations. In the end, using MobX does not need the introduction of other sub-solutions to architect the application state. Using plain MobX plus context is enough to have a good state management solution. However, nothing prevents you to use MobX and Redux (for example) together.

Table 3.1 extrapolates drawbacks and advantages that came out of this conceptual overview. Advantages and disadvantages can sum up.

Table 3.1: Advantages and disadvantages of state management sub-solutions

Approach	Theoretical advantages	Theorical disadvantages
State objects	• No boilerplate • Easy to use	• No SoC • Not suited for complex app
Redux	• Predictable + + • SoC	• No encapsulation • Boilerplate (mostly cause actions-based) • Normalization for big apps
BLoC	• Predictable + • Reusable/Interchangeable • SoC	• Hard to implement • Boilerplate • Hard to use due to streams
MobX	• Easy to use • Less boilerplate • Optimized by default • SoC	• Not very predictable due to implicit and direct mutations
Observer components	• Easy to implement	• Requires effort to handle subscription • Effort to change logic
Stream components	• Reusability • Interchangeable logic	• Hard/tricky because of asynchronous behaviour • Boilerplate
setState	• Easy to use	• Potential unexpected behaviour due to asynchronous process • Bad “performance” due to impossibility of partial rendering (applies mostly on Flutter)

We can now build state management solutions and stipulate a sort of conceptual prediction

of their future behaviour. For example:

- **Solution 1** (using context + state objects + setState) should be:
  Easy to use (double) and boilerplate free but also not suited for complex applications. Not providing SoC benefits neither predictability nor good performances.
- **Solution 4** (using context + BLoC + Stream components) should be:
  Rich of boilerplate (double) because of streams, actions-based mutations. Hard to develop and to use due to the strongly asynchronous workflow. It should also be quite predictable due to action-based mutations and benefit of the SOC advantages. Moreover, the code should be interchangeable and reusable.

In the next chapter I will present three complete state management solutions and their implementation in the Flutter framework.

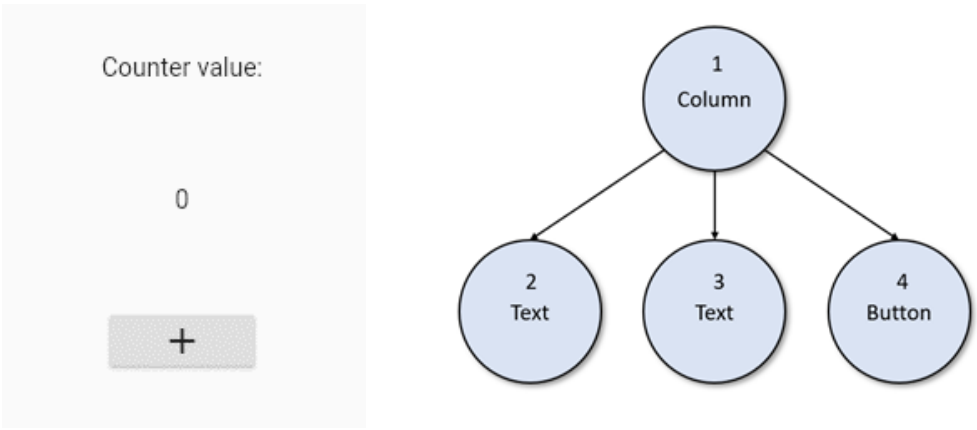
4 | A practical perspective

I will start by proposing a number of practical examples handled with plain setState and state objects in the Flutter framework to highlight all the issues this approach introduces. After that, I will solve the same practical examples with InheritedWidgets (using context) to show how they mitigate the issues introduced by setState. I will continue by showing how the observer and stream components are implemented in Flutter. Lastly, I shall analyse the complete architecture of three state management solutions:

- Redux + using context + stream components
- Bloc + using context + stream component
- Mobx (which uses observer components) + using context

4.1. First common use case

Imagine a simple counter application with a Column containing two Text widgets and a Button. The first Text widget displays a constant text, the second one, the counter value. The Button increments the counter value by one. The (simplified) tree structure looks like figure 4.1a(Widgets have an identification number), whereas its visualization looks like figure 4.1b.



(a) First common use case UI (b) First common use case widget tree

Figure 4.1: First common use case

4.2. Second common use case

Imagine having a UI layered as in figure 4.2.

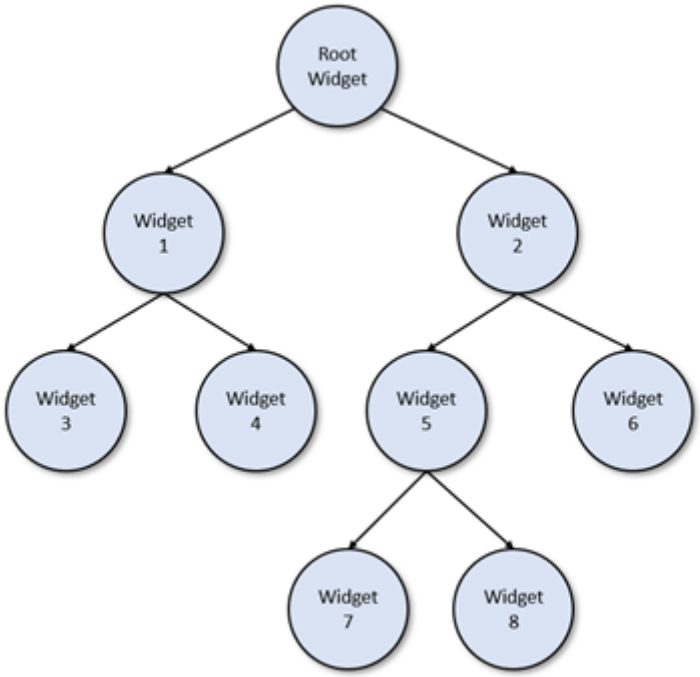


Figure 4.2: Second common use case widget tree

The root widget and its two children (widget 1 and 2) are ruled by a state and rendered

based on it. We want to change their state from other widgets positioned at the bottom of the tree (widgets can automatically change their state if they have one).

In particular:

- Widget 3 needs to change Widget 1 and the Root Widget
- Widget 4 needs to change Widget 1 and the Root Widget
- Widget 5 needs to change Widget 2 and the Root Widget
- Widget 6 needs to change Widget 2 and the Root Widget
- Widget 7 needs to change Widget 2 and the Root Widget
- Widget 8 needs to change the Root Widget

4.3. Managing a state with plain setState and state objects

A setState is the base mechanism provided by Flutter framework to handle dynamic state in a sort of Reactive programming way. It must be used in concomitance with the state object of a stateful widget. Here is an example of a stateful widget handling a counter.

Source Code 4.1: Handling the state of a counter with *setState* and state objects

```
class Counter extends StatefulWidget {
  const Counter({Key? key) : super(key: key);

  @override
  _CounterState createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  // state of the counter
  int counter=0;

  void increment(){
  //call setState with a state mutating function as payload
```

```
    setState(() {
     counter++;
   );
         }
  }
  @override
  Widget build(BuildContext context) {
  // visualize the current counter value
    return Text(counter.toString());
  }
}
```

Notice that the value of the counter is contained in a state object (counter variable) and is equal to 0. SetState comes as a protected method of the State class. The *increment* function implicitly calls setState passing the state changing function as callback.

Let's add a little bit of complexity using the first use case example. (see section 4.2)

In this case, both widget 3 and widget 4 need to access the counter state so it must be placed in the nearest common ancestor; widget 1. The counter value (0) is passed to widget 3 and the *increment* function to widget 4. The widget tree looks like this figure 4.3.

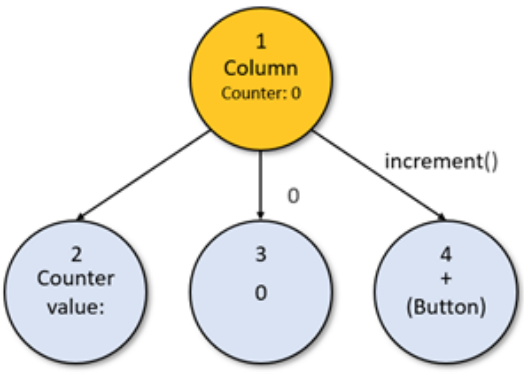


Figure 4.3: First common use case widget tree with *setState*

When the Button is pressed the procedure in sequence diagram 4.4 gets triggered.

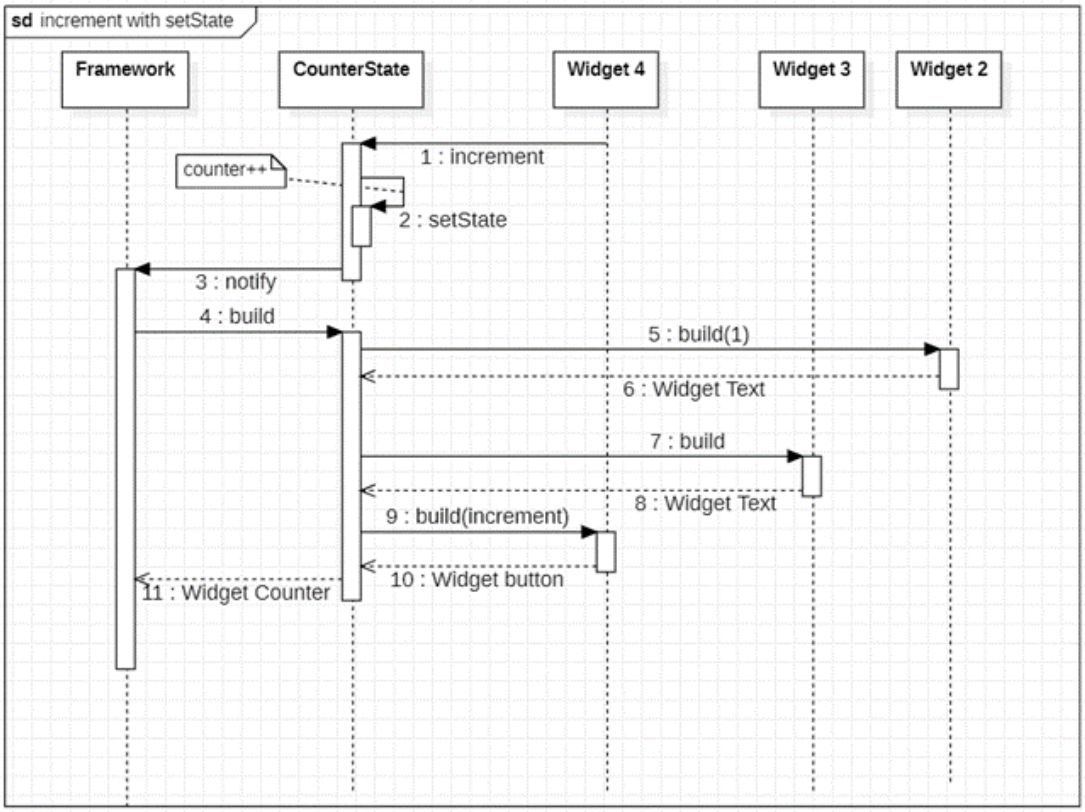


Figure 4.4: Incrementing the counter with *setState*

Notice that, after step 4, all the children's build method gets called even if widget 2 and widget 4 do not change.

What I want to underline with this example is that calling a *build* method leads all the dynamic (not constant) widget in the sub-tree to rebuild. The problem with setState is that it forces all the widgets that access the state not to be constant. Widget 3 and widget 4 cannot be preceded by the *const* keyword because their arguments are not known at compile time. Widget 3 is clearly not constant, it displays a mutating variable, widget 4 is basically receiving a closure that changes over time. The usage of setState requires both the state and its closures to be forwarded to the sub-tree making most of it not markable as constant and, consequently, rebuilt unnecessarily. The ideal scenario would be to have only widget 3 rebuilt but not widget 1 2 and 4; their visualization, indeed, remains unchanged.

Let's see how plain setState behaves with the second common use case. (see section 4.2)

The simplest approach is to wrap the entire state in a single stateful widget, on the root of the tree. Figure 4.5 shows the resulting widget tree.

- Down going arrows forward state. They represent the necessity for the programmer to explicitly wire up parameters forwarding from parent to children.
- State can only be forwarded from father to children. (Unidirectional data flow)
- Different states are forwarded separately. For example, allowing widget 3 to mutate Widget 1 and the root widget requires both their states to be forwarded down the tree.

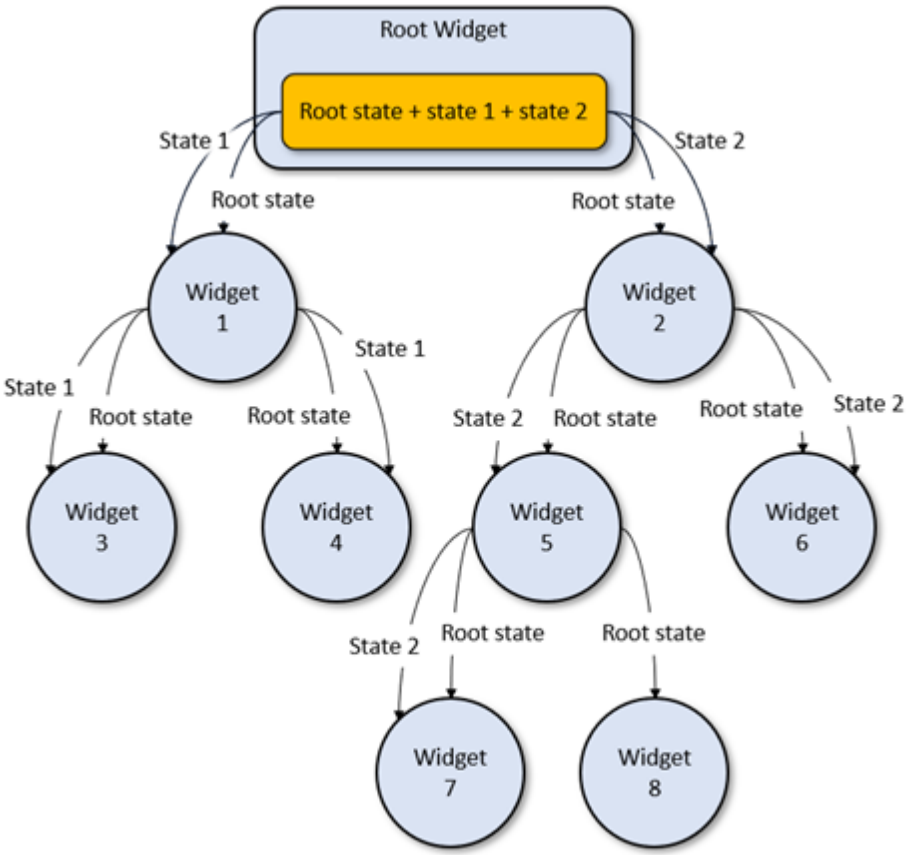


Figure 4.5: Second common use case widget tree with *setState*

Notice that:

- Updating a widget situated **N levels** above requires **N wires** (props drilling problem)
- Updating **N widgets** requires **N wires**

- Every time a state change occurs **all widgets rebuild** (no constant widgets)

Overall, 15 connections are necessary. 15 connections are by far too many for the simple task we are trying to achieve. At every state change 9 widgets rebuild. Moreover, every time the state of widget 1 and 2 changes, the root widget rebuilds. On the other hand, this is the most basic solution possible.

Let's try to do better by moving each state to its corresponding widget (see figure 4.6).

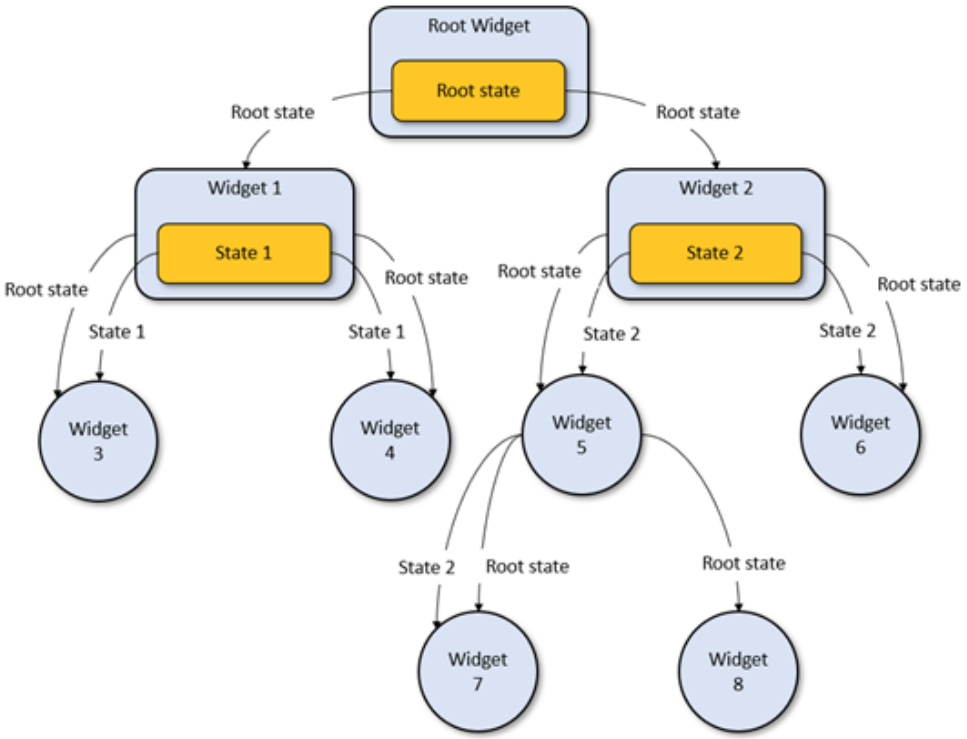


Figure 4.6: Second common use case widget tree with *setState* (optimized)

Notice that:

- Updating a widget situated **N levels** above requires **N wires**
- Updating **N widgets** requires **N wires**
- Every time a state change occurs every widget **in the sub-tree** rebuilds

Move part of the state one level below inside the interested widgets. This cuts down the number of necessary wires to 13. Moreover, if the state of widget 1 or widget 2 changes, the root widget does not rebuild anymore. Precisely, if widget 2 changes, 3 widgets rebuild. If widget 3 changes, 5 widgets rebuild. This is a real improvement; however, this

approach is still not applicable to wider scenarios where the number of widgets exceeds a thousand. Note that, if we want state 2 to be changed in root's left sub-tree, the only possible option is to lift state 2 up to the root widget turning back to situation in figure 4.5.

The setState approach with state objects represents an efficient way of handling state but also suffers from some practical issues:

- does not solve the necessity to provide the state in multiple parts of the application simultaneously and efficiently. Information still needs to be passed Widget by Widget down the tree creating many connections and not enabling constant widgets.
- Causes too many *build* function to be called (due to the problem above)
- The business logic remains tightly coupled with the UI

4.4. InheritedWidget (Using context)

InheritedWidget is a base class provided by Flutter framework. It is used to efficiently propagate information down the tree using context; it also provides a basic implementation of the observer component. An InheritedWidget is a widget that contains information or, in general, a state. Widgets in the Inheritedwidget sub-tree can access the information with fixed cost of O(1). Moreover, a widget accessing an InheritedWidget (called dependent) is automatically rebuilt when the inherited widget changes. More precisely, **only** the accessing widgets in the sub-tree are rebuilt! We will see the benefits introduced by InheritedWidgets later, but first let's see how they are implemented.

To obtain the nearest instance of a particular type of inherited widget from a dependent, use *BuildContext.dependOnInheritedWidgetOfExactType*.

Inherited widgets, when referenced in this way, will cause the consumer to rebuild when the inherited widget changes. Source code 4.2 shows the implementation of a simple InheritedWidget handling a counter.

Source Code 4.2: An InheritedWidget holding a counter value

```
class Counter extends InheritedWidget {
  const Counter ({
    Key? key,
    required this.value,
    required Widget child,}
  ) : super(key: key, child: child);
        //variable containing the current counter value
  final int value;

  static Counter of(BuildContext context) {
    final Counter? result =
     context.dependOnInheritedWidgetOfExactType< Counter >();
    assert(result != null, 'No Counter found in context');
    return result!;
    }
  //if the new value is equal to the
  //previous one do not rebuild dependents
  @override
  bool updateShouldNotify(Counter old) => value != old.value;
}
```

The convention is to provide a static method called *of* on the InheritedWidget which makes the call to *BuildContext.dependOnInheritedWidgetOfExactType*.

The *updateShouldNotify* method determines whether dependents should be informed of a state change.[9]

An InheritedWidget (like the one in Source code 4.2) is a stateless widget and is, therefore, immutable. The state it provides, or better, the snapshot of the state, is injected into it by a stateful widget, which actually contains the mutable state. Source code 4.3 shows how to use a stateful widget to handle the Counter InheritedWidget in source code 4.2.

Source Code 4.3: A stateful widget holding the state of a counter and providing it with an InheritedWidget

```
// the stateful widget containing the actual state
//and providing it to the sub-tree with an InheritedWidget
class CounterProvider extends StatefulWidget {
```

```
  final Widget child;

  const CounterProvider({Key? key, required this.child}) :
   super(key: key);

  @override
  _CounterProviderState createState() => _CounterProviderState();
}

class _CounterProviderState extends State<CounterProvider> {
  //value of the counter
  int value = 0;

  //increment function
  void increment() {
    setState(() {
      value++;
    });
  }

  @override
  Widget build(BuildContext context) {
    //return a screenshot of the state using the Counter InheritedWidget
    return Counter(
        value: value, child: widget.child);
  }
}
```

Let's visualize the entire scenario with an image. Figure 4.7 shows what a widget tree looks like in the first common use case.

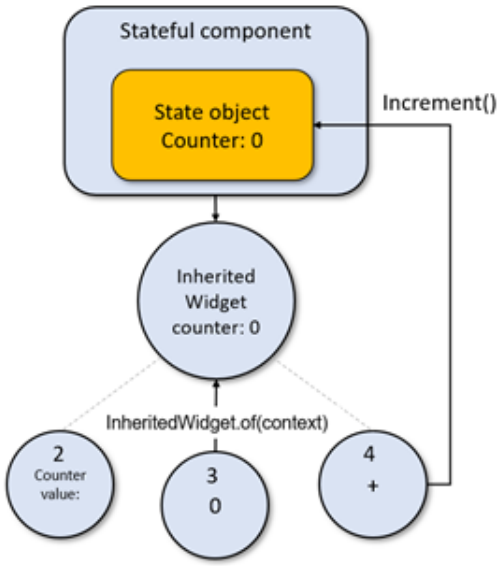


Figure 4.7: First common use case widget tree with InheritedWidgets

The column widget is omitted because it would complicate the diagram without introducing any additional information. In Figure 4.7 a stateful component is holding the counter value. Its child, the Inheritedwidget, provides the actual counter value to widgets 2,3 and 4. Widget 3 accesses the counter value calling the static *"of"* method. Widget 4, once pressed, fires the *increment* function onto the stateful widget starting the procedure exposed by the sequence diagram 4.8.

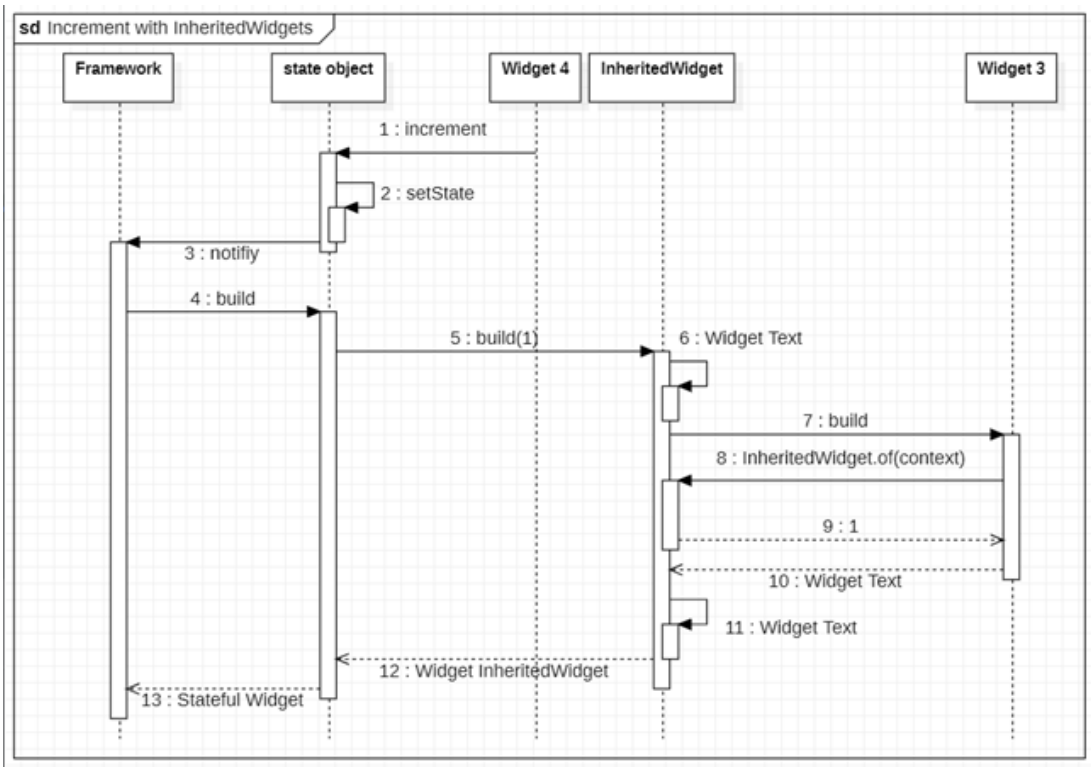


Figure 4.8: Incrementing the counter with InheritedWidgets

The procedure is similar to the one in sequence diagram 4.4 (incrementing a counter with setState) but has some fundamental difference:

- Once the build method of the inheritedWidget is called, the only dependent that sees its *build* method called is the Widget 3 (showing the counter value). This because Widget 3 is the only widget that accesses the counter value through the *of* method (implicitly calling the *dependOnInheritedWidgetOfExactType* method).
- Moreover, Widget 3 is not receiving the counter value from the parent widget anymore, it looks up for the value autonomously in the InheritedWidget.
- Also widget 4 does not access the InheritedWidget at all and, therefore, does not rebuild.

If setState were used, both widgets visualizing and changing the state would rebuild, in this case only the former does.

Let's try to visualize InheritedWidgets benefits with the second use case example. Figure 4.9 shows the resulting widget tree using InheritedWidgets.

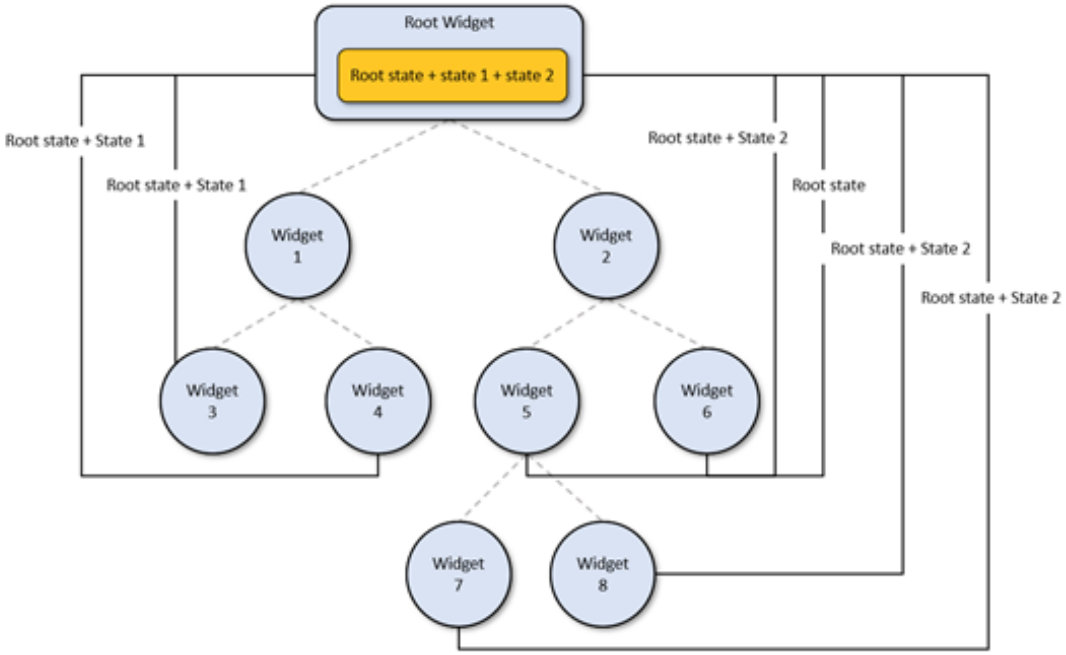


Figure 4.9: Second common use case widget tree with InheritedWidgets

Notice that:

- Updating a widget situated **N level** above requires **one wire**
- Updating **N states** requires **one wire**
- Every time the state changes every widget **accessing the state** is rebuilt

NOTE: wires now point backward to the root widget. This represents the fact that an InheritedWidget is looked up and not dispatched downward.

We cut down the required wires to 6. This is a positive improvement. If we need to access the state from a widget located deep in the tree, a single wire is enough. Notice that we did not specify which widgets access the state, but only listed the ones changing it. If root widget, widget 1 and 2 access the state they only rebuild (3 widgets).

This approach works better with respect to plain setState for complex applications. However, it presents these issues:

- Updating logic is still mixed with the UI. Even if InheritedWidgets implement the observer component pattern, they are not observing an external source of data, they observe a state object. If the state object changes, all the dependents are notified. To be able to use external state container (like Redux and Bloc ones) we need a

more advanced observer component that reacts to an arbitrary external source of state.

- All widgets **accessing** the state are rebuilt on a state change.

I want to propose a third example just to clarify how much the last issue can be tedious. Imagine showing a very long list composed of one hundred items and handling it with InheritedWidgets. As usual, the list is shown with a Column widget (for simplicity) filled with as many items as the length of the list. The state of the list is inserted in the Column widget, thereby accessible to its dependents. The simplified widget tree looks like Figure 4.10.

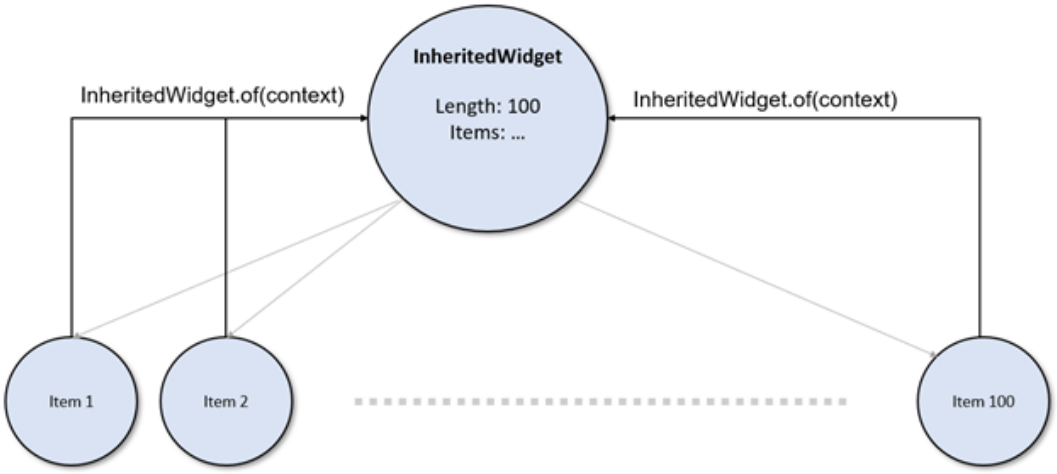


Figure 4.10: Handling a long list visualization with InheritedWidgets

All items access the InheritedWidget to retrieve their own value(s). This means that **changing a single element of the UI leads all the elements to rebuild.** This is clearly not ideal but, at the same time, represents an extreme scenario that should never occur. An application should never show a list of hundreds of items simultaneously. However, Animations, in some particular cases, behave like this. To solve this issue, InheritedModels were introduced.

4.5. InheritedModels

An InheritedModel is an InheritedWidget whose dependents rely on just one part or "aspect" of the overall model.

An InheritedWidget dependent is unconditionally rebuilt when the InheritedWid-

get changes. An InheritedModel is similar to an InheritedWidget with the exception that dependents aren't rebuilt unconditionally. Widgets that depend on an InheritedModel qualify their dependence with a value that indicates which "aspect" of the model they depend on. When the model changes, dependents rebuild only if the aspect they subscribed to changes.

Widgets create a dependency on an InheritedModel with a static method: *InheritedModel.inheritFrom*. Typically, the *inheritFrom* method is called from a model-specific static *"of"* method. For example:

Source Code 4.4: *of* method implementation for an InheritedModel [10]

```
class MyModel extends InheritedModel<String> {
  // ...
  static MyModel of(BuildContext context, String aspect) {
    return InheritedModel.inheritFrom<MyModel>(context, aspect: aspect);
    }
}
```

Calling *MyModel.of(context, 'foo')* means that the widget should only rebuild when the *foo* aspect of the model changes. If the aspect is *null*, then the model supports all aspects.[10]

When the inherited model rebuilds the *updateShouldNotify* and *updateShouldNotifyDependent* methods are used to determine which dependent should be rebuilt. If *updateShouldNotify* returns true, then the inherited model's *updateShouldNotifyDependent* method is tested for each dependent and its corresponding set of aspects. The *updateShouldNotifyDependent* method must compare the set of aspect dependencies with the changes in the model itself as shown in Source code 4.5

Source Code 4.5: An implementation for the *updateShouldNotifyDependent* method [10]

```
class ABModel extends InheritedModel<String> {
  ABModel({ this.a, this.b, Widget child } ) : super(child: child);

  final int a;
  final int b;

  @override
```

```
bool updateShouldNotify(ABModel old) {
  return a != old.a || b != old.b;
 }
@override
bool updateShouldNotifyDependent(ABModel old, Set<String> aspects) {
  return (a != old.a && aspects.contains('a'))
      || (b != old.b && aspects.contains('b'))
        }
// ...
```

In source code 4.5 the dependencies checked by *updateShouldNotifyDependent* are just aspect strings passed to the of method. They are represented as a Set because one Widget can depend on more than one aspect of the model.

InheritedModels solve the issues introduced with the third InheritedWidgets example (see figure 4.10). We can use an InheritedModel (instead of an InheritedWidget) to expose the list and make dependents access the specific item they intend to show. Figure 4.11 visualizes the widget tree when InheritedModels are used.

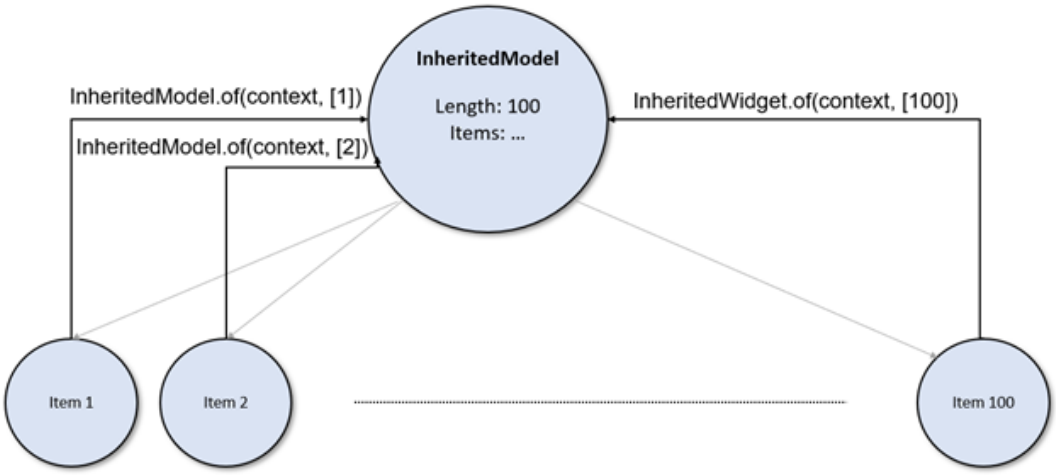


Figure 4.11: Handling a long list with InheritedModels

Item1, for example, accesses the InheritedModel passing its own ID into the aspect field. When the *updateShouldNotifyDependent* method gets called, due to a change in Item3, Item1 is not marked for a rebuild because the change does not affect the aspect it subscribed to.

The wiring in figure 4.11 looks exactly the same as the one in figure 4.10 (about InheritedWidgets) but we have an adjunctive advantage: if a widget changes, **it only rebuilds**. This solves one of two problems related to InheritedWidgets, however, at a cost. The entire logic used to determine which aspect of the model changed and which dependent should rebuild **must be completely defined by the developer** (as seen in the *updateShouldNotifyDependent* method in source code 4.5). This is no minor issue, when the state of the application grows, implementing the updating logic and/or scaling it up becomes exceedingly difficult.

InheritedModel approach goes in the right way but still has two issues:

- The business logic layer is closely coupled with the presentation layer. A dependent still observes a state object, as in the case of InheritedWidgets
- Most of the rendering optimization is left to the programmer even if it could be automatized.

4.6. Provider

The *provider* package by Remi Rousselet and the Flutter team adds the final brick to achieve the complete observer component we are looking for. A Provider widget connects **an arbitrary source** of data (in particular, state) to dependents so that they rebuild as needed to reflect state changes. The external data source communicates with the Provider widget only. The Provider widget then takes care to dispatch data to dependents and to mark them for a rebuild.

A Provider widget is a wrapper around InheritedWidget and InheritedModels that hides their complexity and boilerplate. One could decide to use it just to dispatch arbitrary data to the sub-tree without performing any dependent rebuild, as this would remove the boilerplate introduced by InheritedWidgets.

Let's start visualizing the architecture from a large scale in Figure 4.12.

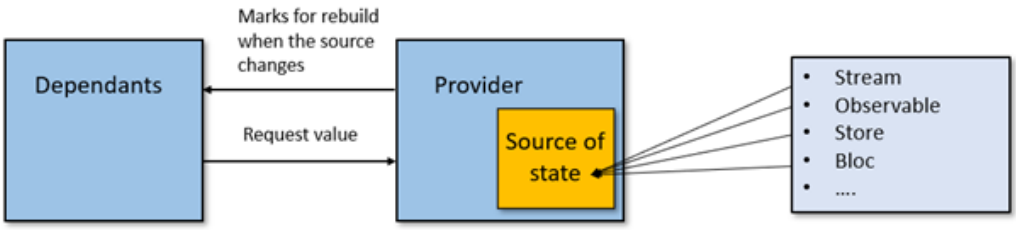


Figure 4.12: Provider widget architecture

Architecture in Figure 4.12 enables the injection of an arbitrary source of state into a Provider widget, which binds it with the UI. This architecture permits a great flexibility. You can choose the external source from a list of possibilities. For example, the provider package supports state sources as *Observable*, *ChangeNotifier*, *Streams*, *Future* and many others.

Let's go ahead with a generic UML diagram that presents the provider architecture at a closer level in Figure 4.13.

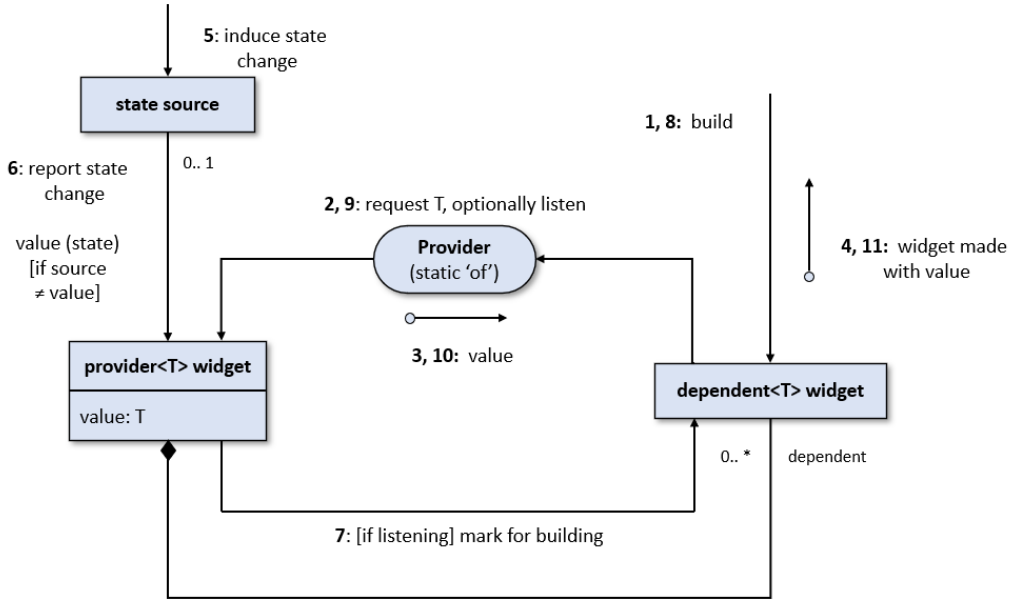


Figure 4.13: Provider architecture at a close level [4]

Diagram 4.13 differs a bit from strict UML to make it more intuitive and more succinct.

Arrows indicate messaging between components. Numbers indicate the messaging order. For example, the notation "1, 8: build" indicates that the 1st and 8th steps are requests to build. Step sequence 5 through 11 may repeat.

Components message each other in the following order, with step numbers corresponding to the numbers in the diagram:

1. Flutter performs the initial build of each dependent widget, prior to any state changes.
2. During the build of a dependent widget, the dependent widget calls *Provider.of<T>()* to request the value it needs, where *T* is the value's type. The widget passes a listen value equal to false if it does not need to rebuild on state changes; otherwise, the widget listens for and rebuilds on changes.
3. The call to *Provider.of<T>()* returns the value (or initial state) of type *T*.
4. The dependent widget finishes building a widget that reflects the value (or initial state) and returns that widget to the Flutter framework for rendering.
5. If the value is dynamically changing state, something induces a state change, such as an external service, a future completing, or a user interacting with a widget.
6. The state source informs the provider of the state change. The state source either delivers the new state value of type T to the provider or the provider retrieves it.
7. The provider, via its InheritedWidget, marks each dependent listening to value type T for building. Doing so induces the dependent to rebuild in Flutter's next rendering frame, which is at most 1/60th of a second later.
8. During the next rendering frame, Flutter tells the dependent widget to rebuild for the new state value.
9. During the rebuild of the dependent widget, the dependent again calls *Provider.of<T>()*, but this time it is to retrieve the new value.
10. The call to *Provider.of<T>()* returns the latest value of type T.
11. The dependent widget finishes building a widget that reflects the new value and returns that widget to the Flutter framework for rendering.

Step 5 through 11 occurs when all of the following conditions are met: (a) the provider has a state source; (b) the provider has a dependent that is listening for state changes; and (c) a state change occurs. Steps 5 and 6 occur on each state change. Step 7 occurs

on each state change when there are dependents. Step 8 through 11 occurs at most once every 1/60th of a second, after a state change, no matter how many state changes occur during this time [6].

4.7. Stream components

Flutter offers a special widget implementing the stream component architecture described in Section 3.6. This component is called StreamBuilder.

To visualize the notion of Stream more easily, simply consider a pipe with 2 ends, with only one end/opening allowing you to insert something into it. When something is inserted into the pipe, it flows out the other end.

In Flutter,

- the pipe is called a Stream;
- to control the Stream, we usually use a StreamController;
- to insert something into the Stream, the StreamController exposes the “entrance", called a *StreamSink*, accessible via the *sink* property;
- the way out of the Stream, is exposed by the StreamController via the *stream* property.

When a components needs to be notified that something is conveyed by a Stream, it has to listen to the *stream* property of the StreamController.

To become a listener, a components uses a StreamSubscription object. This is via the StreamSubscription object which the components is notified that something happens at the level of the Stream [2].

A StreamBuilder listens to a Stream and, each time data comes out that Stream, it automatically rebuilds, invoking its *builder* function/callback. Source code 4.6 shows an example of a StreamBuilder widget handling a counter.

Source Code 4.6: Example of a StreamBuilder widget handling a counter [2]

```
class CounterPage extends StatefulWidget {
  @override
```

```
  _CounterPageState createState() => _CounterPageState();
}
class _CounterPageState extends State<CounterPage> {
  int _counter = 0;
  final StreamController<int> _streamController = StreamController<int>();

  @override
  void dispose(){
    _streamController.close();
    super.dispose();
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(

              body: Center(
        child: StreamBuilder<int>(
          stream: _streamController.stream,
          initialData: _counter,
          builder: (BuildContext context, AsyncSnapshot<int> snapshot){
            return Text('You hit me: \${snapshot.data} times');
          }
        ),
      ),
      floatingActionButton: FloatingActionButton(
        child: const Icon(Icons.add),
        onPressed: (){
          _streamController.sink.add(++_counter);
        },
      ),
    );
 }
}
```

Note that [2]:

- When the FloatingActionButton is tapped, the counter is incremented and sent to the Stream, via the *sink*; the fact of injecting a value in the stream causes the

StreamBuilder, which listens to it, to rebuild;

- The notion of State is no longer needed, everything is taken on board via the Stream;
- This is a big improvement since the fact of calling the *setState* method, forces the whole Widget (and any sub widgets) to rebuild. Here, only the StreamBuilder is rebuilt (and of course its children widgets);
- The only reason a StatefulWidget is being used is simply to release the StreamController via the dispose method.

StreamBuilders are not enough to provide a complete mechanism to make the application react to a unique stream of states. We still have the problem of providing the stream to all widgets listening to it. Fortunately, the introduction of InheritedWidget and Providers targets this specific problem. Provider package comes with some, although the purpose is different, Provider widgets, one of which is called StreamProvider. It listens to a stream and injects the state coming out from the stream into an InheritedWidget, which, rebuilding, leads all descendents (the ones accessing the state) to update. Figure 4.14 shows the architecture of a StreamProvider.

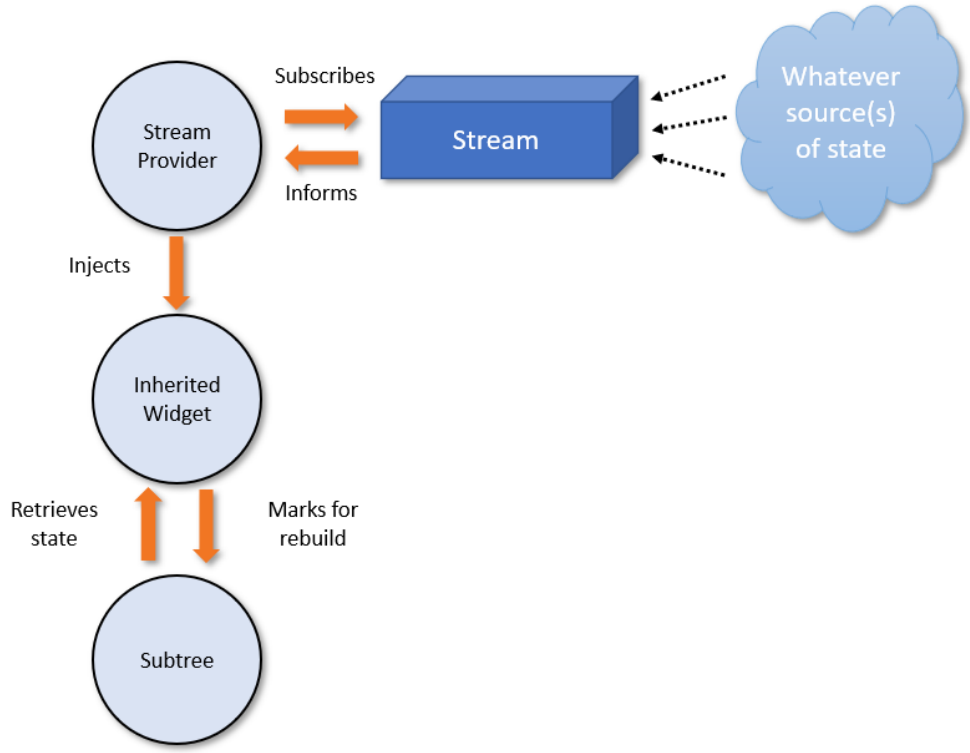


Figure 4.14: StreamProvider architecture

This architecture is really powerful, we can encapsulate our external state container into an object which pushes new state into the stream and use a StreamProvider widget in the visualization layer to have the state automatically synchronized with the UI.

4.8. Redux

Let's introduce the first complete state management solution of the list. The solution: (a) manages the application state using the Redux pattern, (b) dispatches it using context, (c) syncs it with the visualization layer through stream components. We will analyse two packages:

- **redux** (version 5.0.0)
  this package offers an implementation of a Redux store that exposes a stream from where new states pop out. It allows to define a new Store with three parameters: a reducer, an initial state and an optional list of middlewares. It also provides some utilities to reduce the boilerplate generated by reducers and middlewares. (solves question (1))
- **flutter_redux** (0.8.2)
  this package provides a set of utilities to easily consume a Redux Store to build Flutter Widgets. It provides predefined provider and dependent widgets to connect a Store to a UI performing optimizations. (solves questions (2) and (3))

Figure 4.15 describes the architecture of the solution through a component diagram.

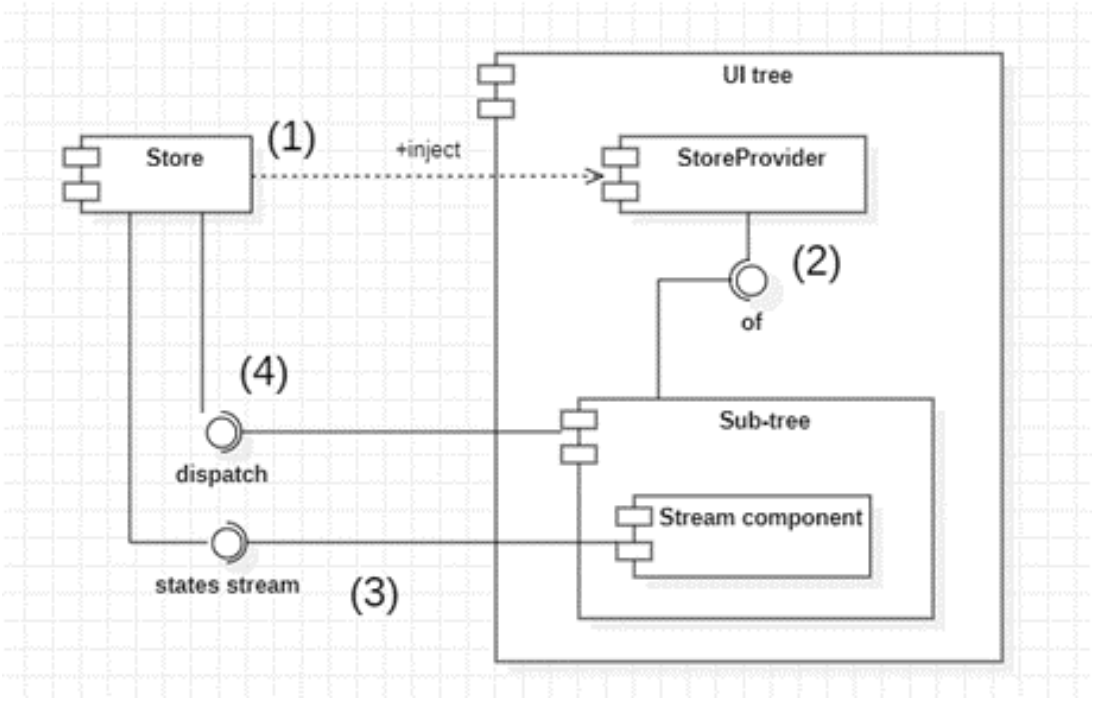


Figure 4.15: Components of an application using a state management solution based on the Redux pattern

Important concepts in the image are marked with numbers:

1. A Store is created with the help of the *redux* package and injected into the UI tree with a StoreProvider,
2. Widgets in the sub-tree retrieve the Store using the static *"of"* method exposed by the StoreProvider,
3. To make a widget listen to changes in the Store it is wrapped into one of the stream components the *redux_flutter* package provides. When a new state pops out from the stream (of the Store) the widget automatically rebuilds,
4. A widget that intends to change the Store dispatches an action using a particular function exposed by the Store. (After performing step 2).

In the following lines I will analyse more in dept how to create a Store with the help of the *redux* package and the internal mechanisms of the stream component from the *flutter_redux* package.

Let's start creating a Store handing a counter. Four things are needed: (1) a model for our application state, (2) a reducer, (3) a set of predefined actions, (4) an

optional list of middlewares. This procedure can be completely defined in pure Dart without using any external library.

Source Code 4.7: Definition of a reducer and two actions

```
class AppState {
  //variable representing the counter value
  int counter = 0;
  // constructor with the possibility
  // to set a value for the counter
  AppState({required this.counter);
}
//action to increment the counter value
class IncrementAction {
  //value to be added
  final int value;
  IncrementAction(this.value);
}
//action to decrement the counter value
class DecrementAction {
  //value to be subtracted
  final int value;
  DecrementAction(this.value);
}

AppState appStateReducer(AppState state, action) {
  //create a new int instance
  //this could be avoided because plain operations
  //return a new instance of the number
  //but this highlights the Redux workflow
  int newValue;
  //check if action is of type increment
  if (action is IncrementAction) {
    //populate the new variable with the actual counter
    //value plus the value contained in the action
    newValue = state.counter + action.value;
    //return a new AppState
    return AppState(counter: newValue);
```

```
        }
  //check if action is of type decrement
  else if (action is DecrementAction) {
    //populate the new variable with the actual counter
    //value minus the value contained in the action
    newValue = state.counter - action.value;
    //return a new AppState
    return AppState(counter: newValue);}
   else {
    //in case the action is unknown return the current state
    return state;}
}
```

Note the determinism of the appStateReducer in Source code 4.7. However, the appStateReducer is hard to read and contains a lot of boilerplate and brackets. Imagine how easy it would be to forget or misplace an ending bracket when the number of actions grows. Source code 4.8 shows the appStateReducer simplified with the combineReducers function and the TypedReducer class provided by the *utils.dart* file of the *redux* package. The code is clearly more readable and scalable.

Source Code 4.8: Definition of a reducer with eased boilerplate

```
final appStateReducer= combineReducers<AppState>([
  TypedReducer<AppState, DecrementAction>(_decrement),
  TypedReducer<AppState, DecrementAction>(_increment),
]);
AppState _decrement(AppState state, DecrementAction action){
  int newValue = state.counter - action.value;
  //return a new AppState
  return AppState(counter: newValue);

AppState _increment(AppState state, IncrementAction action){
  int newValue = state.counter + action.value;
  //return a new AppState
  return AppState(counter: newValue);
```

We can now create a Store object using the appStateReducer and the AppState model we just defined. The Store class is provided by the *redux* package and is instantiated as

shown in Source code 4.9.

Source Code 4.9: Instantiation of a Store object

```
Store<AppState> counterStore=
                Store<AppState>(appStateReducer,
                 initialState: AppState(counter:0));
```

A Store object provides three input/output points: (a) the dispatch public function, (b) a public variable called *state* containing the current state, (c) the endpoint of a stream called *onChange* from which new states pop out.

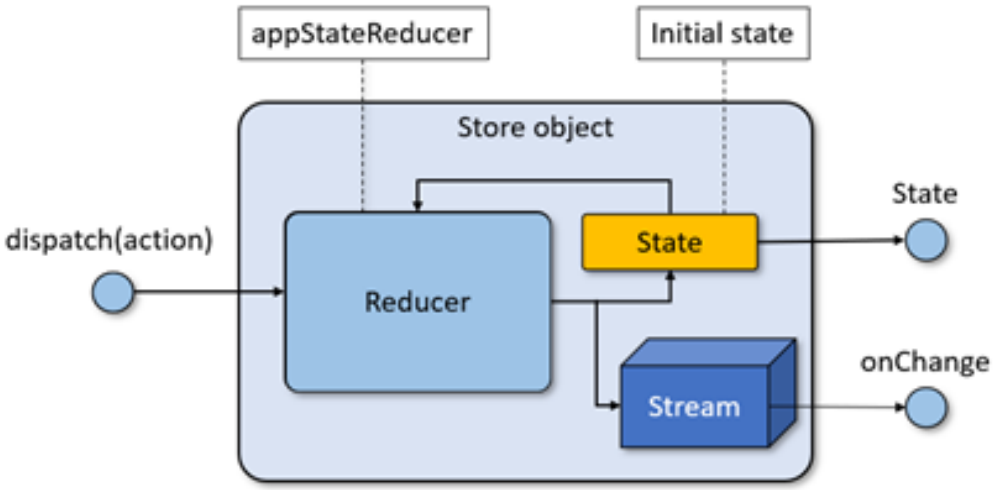


Figure 4.16: Architecture of a Store object

Dotted lines in figure 4.16 represent input values, continuous arrowed lines represent information flow, blue circles represent public access points. I omitted the input list of middlewares to simplify the picture but keep in mind that actions dispatched using the *dispatch* entry point go through every middleware before reaching the reducer.

The dispatch function is used to send an action to the store. The action is processed by the reducer and may produce a new state. The new state is pushed into a stream which endpoint is exposed to the outside and is stored in variable called *state*.

A Store can be bound to a UI with the help of the *flutter_redux* package. A widget called StoreProvider (from the *flutter_redux* package) is used to inject the Store into an InheritedWidget which dispatches it to the sub-tree. The reference to the store object

never changes and neither do the references to its attachment points. For this reason, dependents never rebuild due to a change of the InheritedWidget. Dependents in the sub-tree react to new states coming out from the *onChange* stream using stream components. The *flutter_redux* package provides two of them: the StoreConnector and the StoreBuilder widgets. The difference is that the former can be used to perform optimizations. Both are built on top of a StreamBuilder (introduced in Section 3.6). A StoreConnector attaches to a Store to produce a visual component as shown in figure 4.17.

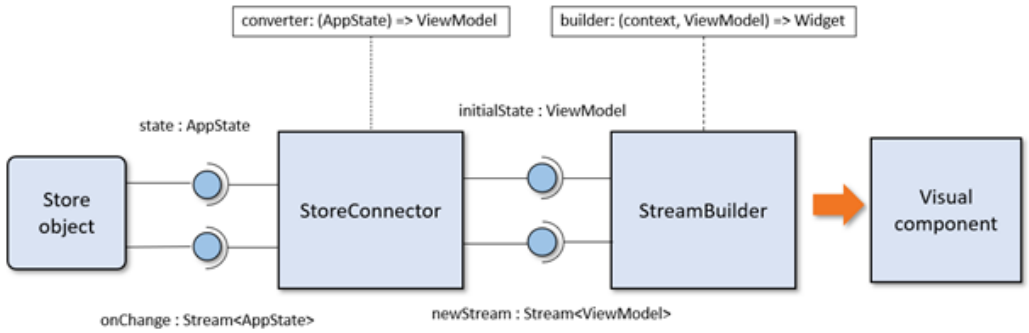


Figure 4.17: Architecture of a StoreConnector linking a Store to its visual representation

A StoreConnector widget is a widget that interacts with the *onChange* stream of a Store. The store is looked up using the StoreProvider's *"of"* method at the StoreConnector creation. The StoreConnector must be provided with a *converter* function that converts an AppState into a ViewModel and a *builder* function that takes a viewmodel and maps it into a visual component (an arbitrary stateless widget). Source code 4.10 shows how to instantiate a StoreConnector widget. In this case, the viewmodel is just an int and represents the actual value of the counter.

Source Code 4.10: Definition of a StoreConnector

```
StoreConnector<AppState,int>(
//converter function takes the store and returns the counter value
    converter: (store)=> store.state.counter,
    builder: (context, counter) {
  // builder function takes the output of the
  // converter function and maps it to a widget
  return Text(counter.toString());
),
```

Let's try to better visualize the process of creating a StoreConnector widget with the sequence diagram 4.18.

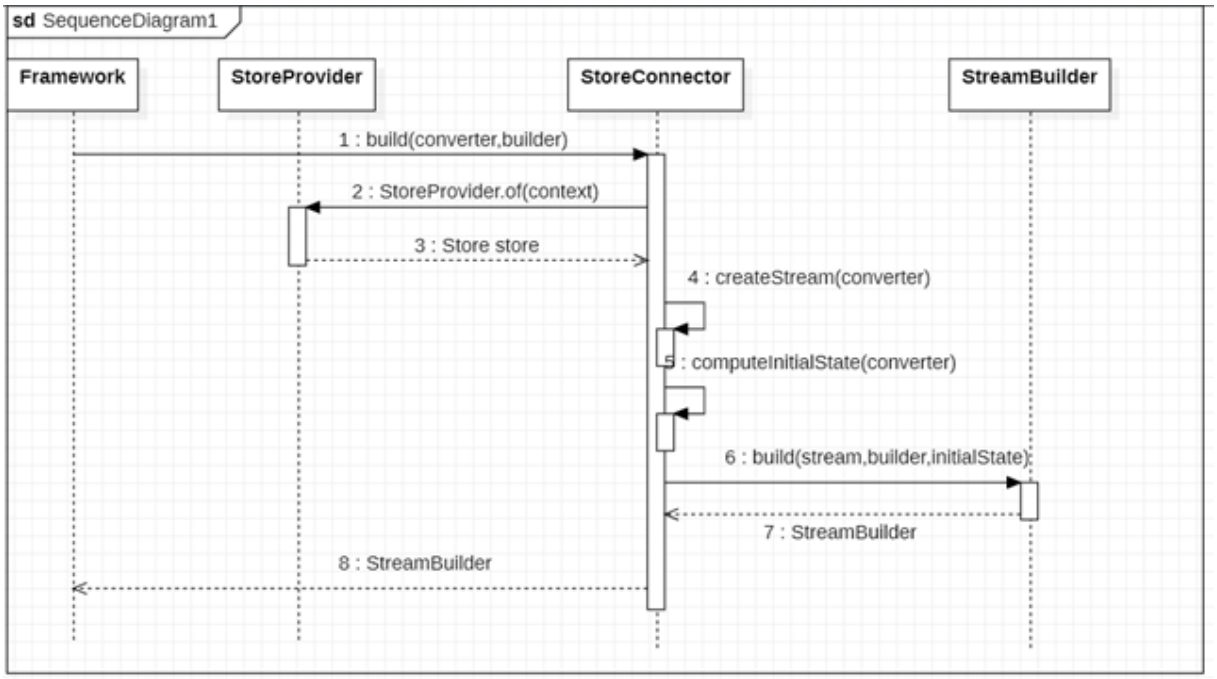


Figure 4.18: Creation of a StoreConnector widget

1. The framework (or the parent widget) instantiates a StoreConnector passing to it: (a) a converter function (mapping an AppState to a viewmodel), (b) a builder function (mapping a viewmodel to a widget).
2. The StoreConnector calls the *"of"* method of the StoreProvider.
3. The StoreProvider returns the instance of the nearest Store.
4. The StoreConnector instantiates a new stream and connects its entry point to the output of the *onChange* stream. This new stream is provided with the converter function and converts AppStates passing through it to viewmodels before they actually reach the StreamBuilder widget.
5. The StoreConnector accesses the current state of the store and converts it to a viewmodel with the converter function.
6. The StoreConnector instantiates a StreamBuilder widget passing to it: (a) the initial state computed at step 5, (b) the stream of viewmodels created at step 4, (c) the builder function obtained from the framework at step 1.
7. The StreamBuilder returns a widget representing the initial state and listening for

viewmodels coming out from the input stream.

8. The StoreConnector terminates its build method execution returning an instance of a StoreConnector widget to the framework.

A StoreConnector is actually a widget that hides the boilerplate and the complexity generated by the process of linking a Store to a StreamBuilder widget.

Let's talk about the objective of the converter function before giving a clarifying example of the overall process. A converter function is a selector that filters the information required by a stream component from the store. When a new state pops out from the stream, the StoreConnector computes a new viewmodel and compares it with the previous one; if they match, it does not rebuild. The advantage of this mechanism is that the stream component just listen to the aspects of the state it is interested in. This allows it to rebuild only when that specific aspect (or aspects) changes. Without a converter function, a component would listen directly to the store and rebuild when a single variable changes. As usual, this behaviour is not ideal, but in this case it is even worse because of the Redux pattern habit to convey the whole state into a single object.

I would like to conclude with a simple example. Imagine having a Todo application composed of three widgets, one showing the entire list of todos, one showing the completed todos and one showing the pending ones. The store contains the complete list of todos and is able to receive actions in order to add, delete, and modify todos. The Store is dispatched to dependents using a StoreProvider. Each widget: retrieves the store from the StoreProvider, computes its own viewmodel and renders it.

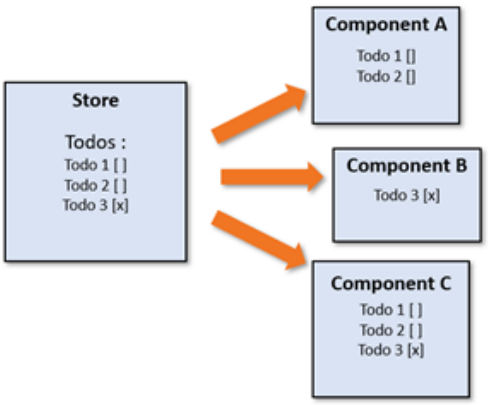


Figure 4.19: Components accessing a Store with their viewmodels

Figure 4.19 shows the situation after every component builds. At some point, the store receives a request (in the form of an action) to add a pending todo to the list, called Todo 4. It processes the action and emits a new AppState in the output stream. Each component receives the new state and computes its own viewmodel rebuilding if the viewmodel differs from the previous one. Component A and C rebuild because their viewmodel changes. Component B does not rebuild because there are no changes in its viewmodel. Figure 4.20 shows the situation after the insertion of Todo 4.

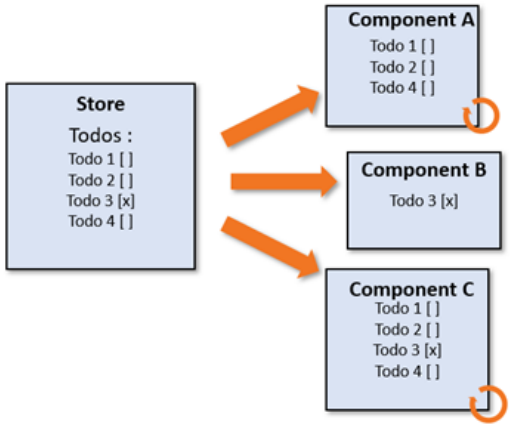


Figure 4.20: Components rebuilding due to a change in the viewmodel

Note the simplicity of this mechanism, once all components are set up to listen to the store, updating the UI is as simple as changing the state. Changes are propagated automatically to the view. Also note that viewmodels must be provided by the developer and come with and overridden equality operator (==) to be comparable. Defining a viewmodel for every component accessing the state introduces some boilerplate and a warning issue. In example 4.19, at every state change, all components compute their own viewmodel. This means that, if an application is composed of a thousand Component B, a thousand of the same viewmodel B are computed at each state change.

We will see the drawbacks of this approach later.

4.9. BLoC

Let's introduce the second complete state management solution of the list. The solution: (a) manages the application state using the BLoC pattern, (b) dispatches it using context, (c) syncs it with the visualization layer through stream components. We will analyse two

packages:

- **bloc** (7.2.1)
  this dart package that provides an implementation of the BLoC pattern and hides great part of the boilerplate.(solves question (1))
- **flutter_bloc** (7.0.0) (solves question (1))
  this package provides a set of utilities to easily consume one or more blocs to build Flutter Widgets. It provides predefined provider and dependent widgets to connect blocs to a UI and perform optimization. (solves questions (2) and (3))
- **equatable**(2.0.0)
  Equatable overrides the equality operator (==) and the hashCode so you don't have to waste time writing lots of boilerplate code to compare objects in Dart language

Figure 4.21 describes the architecture of the solution through a component diagram.

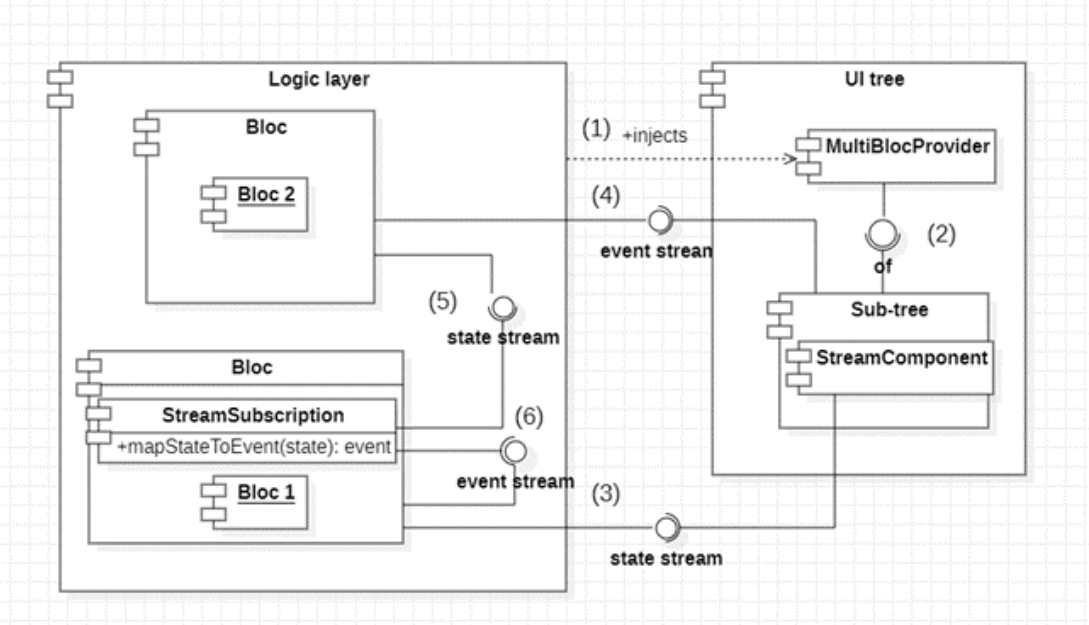


Figure 4.21: Components of an application using a state management solution based on the BLoC pattern, stream components and context propagation

Important concepts are marked with numbers:

1. A logic layer composed of one or more blocs is created with the help of the Bloc class from the *bloc* package and injected into the UI tree with a MultiBlocProvider (or a BlocProvider for a single bloc).

2. Widgets in the sub-tree retrieve a bloc using the static *"of"* method exposed by the MultiBlocProvider.
3. To make a widget listen to changes in a bloc it is wrapped into one of the stream components the *bloc_flutter* package provides. When a new state pops out from the stream (of the bloc) the widget automatically rebuilds.
4. A widget that intends to emit an event in a bloc uses the event stream sink exposed by the bloc. (After performing step 2)
5. A bloc (Bloc 1) subscribes to states coming out of the state stream of another bloc (Bloc 2) with a StreamSubscription.
6. The StreamSubscription maps the new emitted state to zero, one or more events and emits it/them in the event stream sink of the bloc (Bloc 1). The bloc may produce one or more new states firing the procedure of step 3.

In the following lines I will analyse more in dept how to create a bloc with the help of the *bloc* package and how to bind it to a stream component from the *flutter_ bloc* package.

Let's start with an implementation of a bloc in plain Dart code. A bloc is an object containing two streams, one for input events and one for output states, and a private method that consumes events to produce new states, called *mapEventToState* as convention.

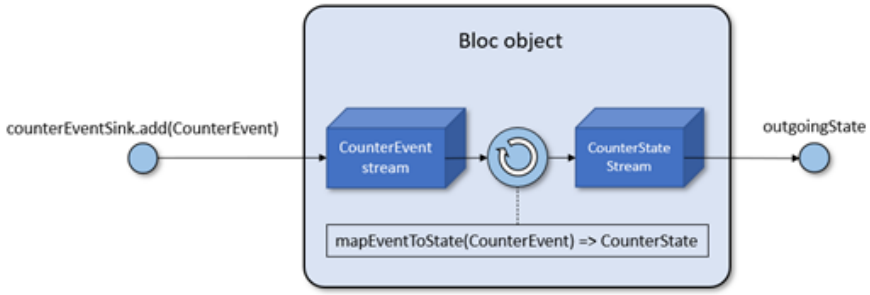


Figure 4.22: Architecture of a bloc object

Figure 4.22 shows the architecture of a bloc object. It exposes one input variable, the *sink* of an event stream, and one output variable, the endpoint of a state stream. Source code 4.11 shows the implementation of a bloc handling a counter in pure Dart code plus the definition of an increment event.

Source Code 4.11: Definition of a bloc handling a counter and an event [2]

```
class CounterBloc {
  int value = 0; // initial state

  // broadcasting stream so it can be used multiple times
  final _controlStateController = StreamController<int>.broadcast();

  StreamSink<int> get _incomingValue => _controlStateController.sink;

  Stream<int> get outgoingState => _controlStateController.stream;



  final StreamController<CounterEvent> _counterEventController =
   StreamController<CounterEvent>.broadcast();

  StreamSink<CounterEvent> get counterEventSink =>
   _counterEventController.sink;

  CounterBloc() {
    _counterEventController.stream.listen(_mapValuesToState);
  }

  void _mapEventsToState(CounterEvent event) {
    if (event is AddEvent) {
      value=value+event.value;
      _incomingValue.add(value);
   }
  }

  void dispose() {
    _controlStateController.close();
    _counterEventController.close();
  }
}
```

```
abstract class CounterEvent {

class AddEvent extends CounterEvent {
  final int value;

  AddEvent(this.value);
}
```

Note:

- usually, blocs take as input the initial state; in this case the initial state is hardcoded in the CounterBloc
- controlStateController is a StreamController that handles the stream of states
- counterEventController is a StreamController that handles the stream of events
- the only public variables are the outgoingState stream and the counterEventSink sink.

Source code 4.11 contains a great amount of boilerplate. Let's define a bloc with the help of the bloc package in a more concise way in Source code 4.12.

Source Code 4.12: Definition of a bloc hadling a counter with eased boilerplate

```
class CounterBloc extends Bloc<CounterEvent, int> {
  /// The initial state of the `CounterBloc` is 0.
  CounterBloc() : super(0);

  /// and a new state is emitted via `emit`.
  @override
  Stream<int> mapEventToState(CounterEvent event) async* {
    /// When a `CounterIncrement` event is added,
    if (event is CounterIncrement) {
      /// the current `state` of the bloc is accessed via the
      //  `state` property and a new state is emitted
      yield state + event.value;
    }
  }
}
```

Source code 4.12 is clearly more readable, boilerplate-free and maintainable.

Blocs are provided to dependents using a widget called BlocProvider, offered by the *flutter_bloc* package. A BlocProvider is a wrapper around InheritedWidgets.
A dependent looks up a bloc using the *"of"* method of the BlocProvider widget. A dependent reacts to state coming out of the bloc state stream using a stream component. Stream components are provided by the *flutter_bloc* package in different types: BlocBuilder, BlocListener, BlocConsumer and BlocSelector.

The BlocBuilder is the most general stream component; it takes the state coming out of a stream and converts it into a widget.

The BlocListener reacts to states coming out of a stream with side effects, such as showing a snackbar or navigation.

The BlocSelector is an advanced version of the BlocBuilder that extrapolates a view-model from the current state and uses it to determine whether to rebuild. A BlocSelector behaves like a StoreConnector in Redux, and the principle behind them is exactly the same. StoreConnectors use StreamBuilder widgets under the hood whereas, BlocSelector widgets are built from scratch by the *flutter_bloc* package and use *setState*.

Let's see how to declare a BlocBuilder listening to the CounterBloc defined in source code 4.13.

Source Code 4.13: Definition of a BlocBuilder widget

```
BlocBuilder<CounterBloc, int>(
builder: (context, counterValue) {
  //access the state in the builder and print the counter value
  return Text(counterValue.toString());
  }
),
```

NOTE: in Source code 4.13 the counter state is represented by an int variable. In more complex applications, states usually are wrapped into predefined objects (for example LoadedCounterState). In this context, the *equatable* package is really handy. It automatically performs equality operator and hashcode overrides. Source code 4.14 shows how to define comparable states using the *equatable* package in the context of a todos application.

Source Code 4.14: Usage of the Equatable package

```
abstract class TodosState extends Equatable {
  const TodosState();

  @override
  List<Object> get props => [];
}
class TodosLoadedState extends TodosState {
  final List<Todo> todos;

  const TodosLoadedState(this.todos);

  @override
  List<Object> get props => [todos];
}
void main(){
  TodosLoadedState state= const TodosLoadedState([]);
  print(state == TodosLoadedState([])); //true
}
```

If TodosState class did not extend the Equatable interface, the comparison would return false.

4.10. MobX

Let's introduce the third complete state management solution of the list. The solution: (a) manages the application state using the MobX reactivity system, (b) dispatches it using context, (c) syncs it with the visualization layer through observer components. We will analyse two packages:

- **mobx** (2.0.5)
  this package implements the core of MobX. It provides all the base classes to create a reactive context. Mobx, in fact, uses an ad hoc created context to perform all the optimizations and automatic subscriptions. This package is quite dense. (solves question (1))
- **flutter_mobx** (2.0.2)
  this package provides the Observer widget that listens to observables and automati-

cally rebuilds on changes. This package and the previous one are strongly dependent. Redux and BLoC core packages can be used without prebuilt widgets, whereas, the MobX core must be integrated with this package to work properly. (solves question (2))

- **provider** (6.0.1)
  this package has been introduced in Section 4.6. It is used in this state management solution to dispatch the application state to dependents (solves question (3)).
- **build_runner**(2.1.4) and **mobx_codegen**(2.0.4)
  These two packages provide a code generator used by MobX to offer a boilerplate free solution.

I would like to describe this state management solution directly with a practical example. MobX reactivity system and architecture are, in fact, quite dense and hard to understand. I think this is coherent with MobX objective to remove all the complexity, the effort and the boilerplate of setting up a reactivity system and to jump straight to the core of the implementation process. An utilizer is not supposed to understand its internal mechanisms in depth.

Take the first common use case exposed in Section 4.1. Instead of showing the counter value as it is, we want to show it doubled. Source code 4.15 creates a class to contain the counter value and an action to increment it using the *mobx* package.

Source Code 4.15: A store handling a counter with MobX

```
// Include generated file
part 'counter.g.dart';

// This is the class used by rest of your codebase
class Counter = _Counter with _\$Counter;

// The store-class
abstract class _Counter with Store {
  @observable
  int value = 0;

  @action
  void increment() {
    value++;
```

```
  }
}
```

The interesting parts are:

- The abstract class _Counter that includes the Store mixin that provides the *@observable* and *@action* annotations,
- The *counter.g.dart* file included with the *part* directive. This file contains the generated code. Without this line the *build_runner* would not produce any output. The generated file contains the *_$Counter* mixin.
- The *@observable* annotation to mark the counter value as an observable.
- Use of *@action* to mark the *increment* method as an action.

I will leverage the MobX reactivity system to compute the doubling of the counter value. To do so a computed value is declared. It will be synced to the observable state it depends on by MobX, automatically. To define a computed value, it is enough to mark it with the *@computed* annotation as shown in Source code 4.16.

Source Code 4.16: Definition of a computed value

```
@computed
int get double {
  return _value * 2;
}
```

At this point the code in *counter.g.dart* can be generated using the following command in a terminal set to the project folder.

flutter pub run build_runner build

This concludes the MobX store definition, let's now connect it to the UI.

The store is dispatched to dependents with a Provider widget (from the *provider* package). Dependents retrieve the store using the *"of"* static method of the Provider class.

To connect an observable in the store to the UI, the *flutter_mobx* package provides a prebuild observer widget called Observer.
An Observer widget takes as argument a builder function. The builder function returns

an arbitrary widget using one or more observables retrieved from the store. When an observable value changes, the Observer widget is automatically rebuilt.
Source code 4.17 shows an Observer widget using the computed value defined in Source code 4.16 and a FloatingActionButton incrementing the counter value.

Source Code 4.17: Accessing and updating a MobX store

```
@override
Widget build(BuildContext context) {
  //retrieve the counter using the of method
  final Counter counter = Provider.of<Counter>(context);
  return Scaffold(
      // wrap the Text widget inside a Observer widget to
      // automatically rebuild after a state change
      body: Observer(builder: (context) {
        //access the counter value
        return Text(counter.double.toString());
        }
      ),
      floatingActionButton: FloatingActionButton(
        //increment the counter value
        onPressed: ()=>counter.increment(1),
      ),
  );
}
```

Note that an Observer widget is not provided with any particular store or bloc, it is not typed nor provided with any viewmodel, it makes widgets in its sub-tree reactive to changes in the application state automatically without any wiring. It is as flexible and interchangeable as a stream component but also boilerplate-free as an observable one. It is like using a stream component but keeping the benefits of an observable.

5 | The Todo app

This chapter describes the implementation of a mobile application handling a list of todos. The application is developed once for each of this three state management solutions:

- BLoC + Stream components + context
- Redux + Stream components + context
- MobX + context

Each implementation is divided in two sub-processes:

- Implementation of the base functionalities
- Renderings optimization

The end of each subsection reports a summary of the data collected during the implementation. Collected data refers to the lines of codes produced and the time spent at each sub-process.

5.1. General overview

This section describes the output of each sub-process.

5.1.1. Base functionalities

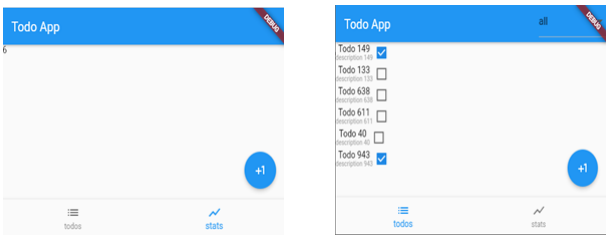
The output application of this sub-process partially handles a list of todos.

It offers the possibility to:

- Visualize a list of todos with their names, descriptions, and competitions,
- Filter the list based on a filter (completed, pending or both),
- Visualize statistics about the completed todos.

The output application is composed of a single page, called the HomePage. The HomePage

varies based on the value of a tab variable. When the tab value is set to *"todos"* (see Figure 5.1a) the HomePage shows the list of todos and provides a DropdownButton to filter them. When the tab is set to *"stats"* (see Figure 5.1b) the HomePage shows a numerical summary of the current todos situation.



(a) *todos* tab UI (b) *stats* tab UI.

Figure 5.1: HomePage UI

5.1.2. Renders optimization

This step aims at minimizing the number of widgets rebuild at every state change. It mostly targets the problem introduced in Section 4.4 with Figure 4.10, when a list with an arbitrary number of elements is displayed. In short, without any optimization, the entire list of todos rebuilds when one of the elements changes, worsening performances and memory consumption.

5.2. Implementation

5.2.1. Shared project structure and files

This subsection presents the parts of the code shared between different implementations.

Models - Let's start defining the models for the application. Class diagram 5.2 shows the classes used to build the application state.

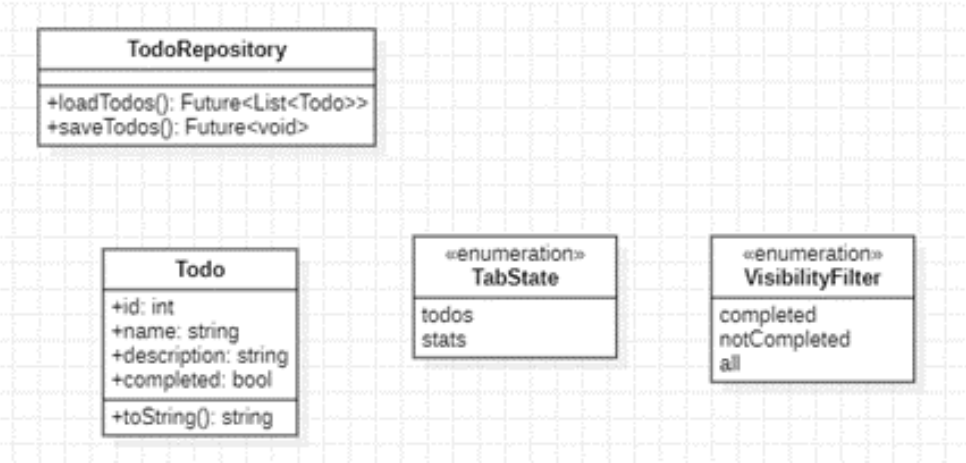


Figure 5.2: Shared models (Todos app)

The TodoRepository class simulates the retrieval of a list of todos from a database. The *loadTodos* and *saveTodos* methods are asynchronous and produce a delay of 2 seconds to simulate the retrieval process. The *loadTodos* method returns a list containing six todos with random ids.

User Interface - The special purpose widgets used to build the three main pages are:

- The TodoItem widget. It displays information about a specific todo and provides a checkbox to change its competition field.
- The TodoView widget. It is a special purpose widget that takes a list of todo entities and displays it. It maps the list to a list of TodoItem widgets.
- The VisibilityFilterSelector widget. It is a special purpose widget that allows to swap between filters using a DropdownButton.
- The TabSelector widget. It is a special purpose widget that enables to swap between tabs using a BottomNavigationBar.

5.2.2. Implementation based on InheritedWidget/Model and SetState

This Section describes the implementation and the architecture of the todo application (see Section 5.1) using the base state management tools offered by the Flutter framework. The solution in question uses: (a) state objects to contain state, (b) context to dispatch it and (c) observer components to kept UI synchronized. In particular, state dispatchment and dependents rebuilding are performed using InheritedWidgets.

InheritedWidgets require the state to reside in a stateful widget. The stateful widget injects data in a InheritedWidget and makes it accessible in the sub-tree. In this implementation, the stateful widget is named TodoProvider, the InheritedWidget TodoInheritedData.

I treated the tab value as an ephemeral state and the rest of the application state as a shared state. In particular the shared state contains:

- The list of todos
- The filtered list of todos
- The current visibility filter
- The stats

Let's start to visualize the architecture of the application with Figure 5.3.

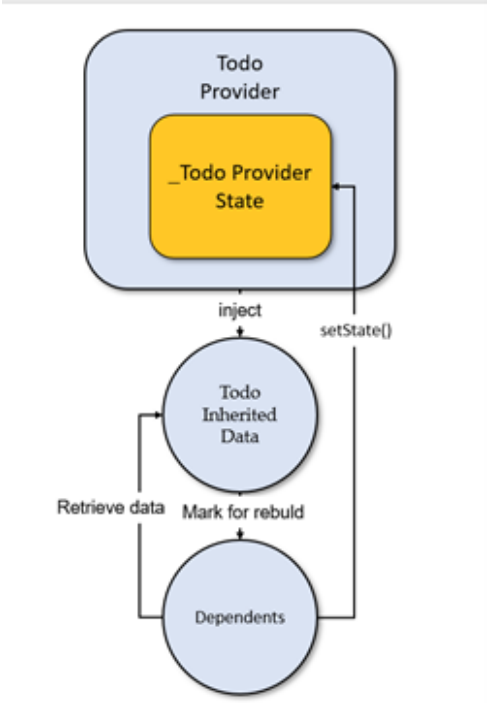


Figure 5.3: Architecture of the application based on InheritedWidgets

This architecture has been explained in Section 4.4. After the UI is built, the workflow is the following:

- A dependent (or the TodoProvider widget itself) updates the _TodoProviderState

with *setState*;

- TodoInheritedData widget rebuilds with the a new snapshot of the state;
- TodoInheritedData widget marks all listening widgets in the subtree for a rebuild;
- Dependents marked for a rebuild start the build process;
- Rebuilding dependents access the TodoInheritedData widget using the static *"of"* method to retrieve data they need
- Dependents terminate the building process and the UI is updated

Class diagram 5.4 shows a more accurate representation of the TodoProvider widget and the TodoInheritedData widget.

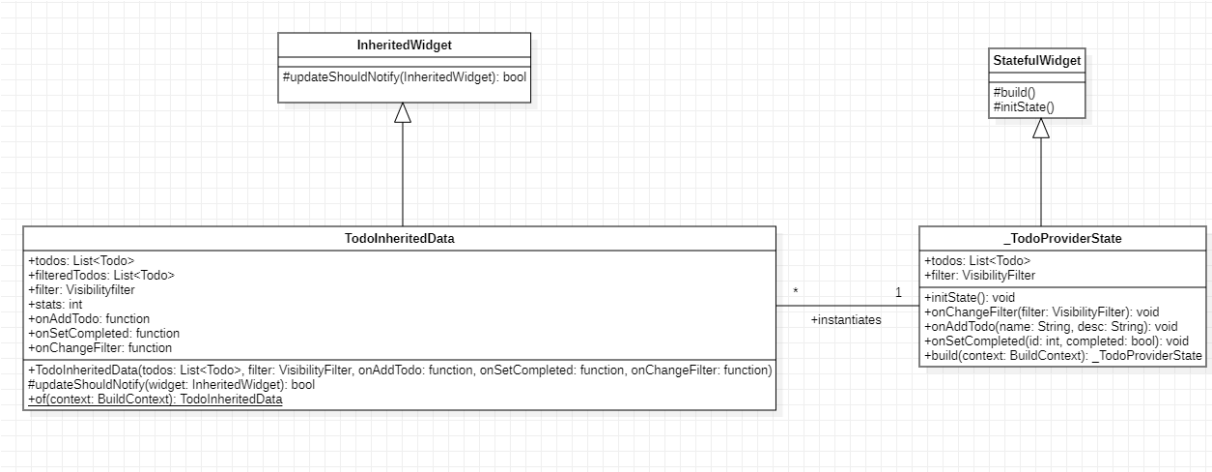


Figure 5.4: TodoInheritedData class extending the InheritedWidget interface

Note that:

- _TodoProviderState holds the actual list of todos and the current visibility filter
- *onChangeFilter*, *onAddTodo* and *onSetCompleted* are defined in the _TodoProviderState and implicitly call the *setState* method to update the list and the filter;
- TodoInheritedData constructor receives a reference of the state changing functions from the _TodoProviderState. Dependents can access these functions directly through the TodoInheritedData instead of receiving them from the parent widget (props drilling)
- TodoInheritedData constructor just receives the list of todos and the filter. The filtered list and the stats are calculated later;

- The *updateShouldNotify* method compares the current list and the current filter with their old values, in case they are both equal, it returns *false* avoiding dependents to rebuild;
- The _TodoProviderState *initState* method executes before the widget builds and contains the actual fetching of the list.

Optimize renderings was a pretty hard task. I spent hours trying to make the single TodoItem rebuild instead of the entire TodoView before realizing that it was just unfeasible using plain InheritedWidgets. Searching for a solution, I came across InheritedModels that target this exact use case. InheritedModels are introduced in Section 4.5.

Class diagram 5.5 shows the updated TodoInheritedData class.

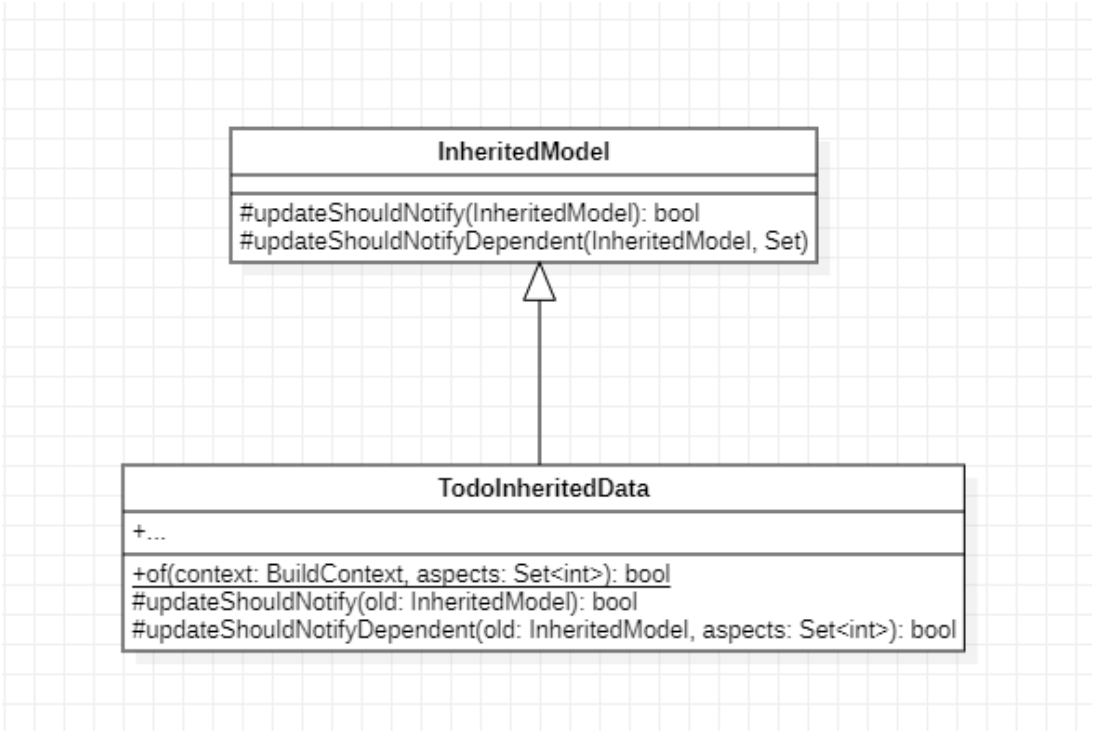


Figure 5.5: TodoInheritedData class extending the InheritedModel interface

To leverage this mechanism, I set up a mapping from int numbers to model aspects. I represented with the number:

- **0**: a change that affects the entire list of todos. For example, adding a todo or deleting a todo. This kind of change require the entire TodoView to rebuild. (no todo with id 0 for convention)
- **N** (where N is the id of the todo): a change that affect only the todo with id N.

The TodoView accesses the InheritedModel passing the number 0 in the aspect parameter, whereas the single TodoItem accesses the InheritedModel passing the id of its todo.

The last thing to do is to override the *updateShouldNotifyDependent* function implementing the logic through which changes in the model are bound to aspects.

This function was pretty hard to code, Source code 5.1 reports the pseudocode.

Source Code 5.1: *updateShouldNotifyDependent* method pseudocode

```
@override
bool updateShouldNotifyDependent() {
  if (changeAffectingTheEntireListOccured) {
    //leads every dependent to rebuild whatever aspect it subscribed to
    return true;
  } else {
    // in case the change is not affecting the entire TodoView
    //check which aspect the dependent subscribed to
    if (dependencies.contains(0)) {
      //if it subscribed to structural changes
      //do not rebuild because the change is not structural
      return false;
    }
    if (todoWithID(Dependencies).changed) {
      //if the todo with id equal to the value in the dependencies changed
      // evaluate to true (rebuild)
      return true;
    }
  }
  //if no previous statements were satisfied return false
  return false;
}
```

Table 5.1 shows a summary of the collected data during the implementation process.

	lines of code	time	lines/time ratio	classes
base functionalities	507	2-3 h	2.81 l/m	2
rendering optimization	45	8-10 h	0.084 l/m	0

Table 5.1: Collected data during the implementation process based InheritedWidgets (Todos app)

5.2.3. Implementation based on Redux

This section describes the implementation and the architecture of the *todos* application (see Section 5.1) using a state management solution based of Redux. The solution uses: (a) the Redux pattern to handle the shared application state, (b) context to dispatch it and (c) stream components to kept UI synchronized. The implementation uses two Flutter packages: *redux*(version) and *flutter_ redux*(version) (see Section 4.8).

I treated the entire application state as a shared state. In particular, the shared state contains:

- The list of todos
- The current visibility filter
- The current tab value

Let's start to define the elements of the Redux store. Class diagram 5.6 describes the model for the application state plus the models for the actions.

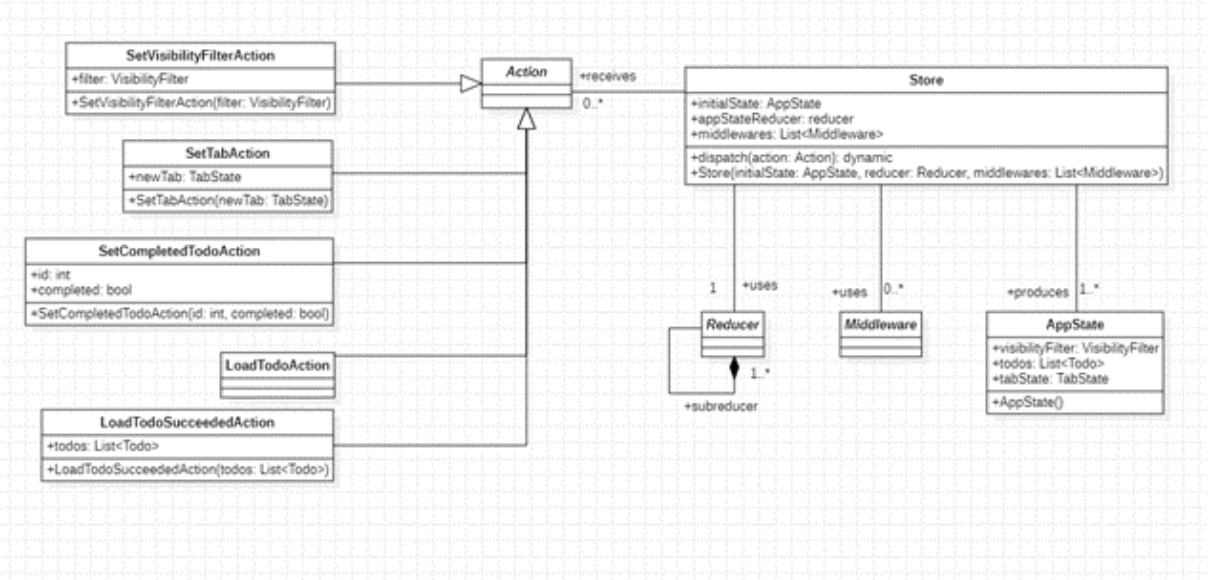


Figure 5.6: Class diagram regarding the logic layer in the implementation based on the Redux pattern

Note:

- LoadTodoAction starts the fetching process;
- LoadTodoSucceededAction ends the fetching process and contains the fetched list;
- The Store class is provided by the *redux* package and exposes a *dispatch* function taking a generic action as single argument;
- I omitted other general-purpose functions from the Store class definition for simplicity;
- The constructor of the AppState does not take any argument; the list is set to be empty, the filter is set to "all" and the tab is set to *"todos"* by default.

The application uses a single middleware, called *loadTodosMiddleware*. Its implementation is shown in Source code 5.2.

Source Code 5.2: A middleware fetching a list of todos from a repository

```
void loadTodosMiddleware(Store<AppState> store, action,
 NextDispatcher next) {

  if (action is LoadTodoAction) {
    TodoRepository.loadTodos().then((todos)
```

```
      {store.dispatch(LoadTodoSucceededAction(todos));} );
  }
  next(action);
}
```

The middleware is used to handle the fetching process. A middleware is just a function of type *void* taking three parameters: the store, the action and the next middleware. The *loadTodosMiddleware* middleware intercepts actions before they reach the *appStateReducer*. The NextDispatcher is the next middleware in the list or, if no other middleware is present, the *appStateReducer*. The *loadTodosMiddleware* reacts to actions of type LoadTodoAction. Before passing the action to the *appStateRecuder* it starts the fetching process. It also takes care to dispatch an action of type LoadTodoSucceededAction when the fetching process ends.

The application uses a StoreProvider widget situated on top of the widget tree. Several StoreConnector widgets access the Store using the *"of"* method and listen to states coming out of the *onChange* stream. Widgets that intend to mutate the AppState call the *dispatch* method with an action as payload.

Widgets listening to the Store with their corresponding viewmodels are:

- The HomePage which depends on the tab value (TabState)
- The TodoView which depends on the list and on the filter (List<Todo>)
- The Stat which depends on the list of todo (int)
- The TabSelector which depends on the tab value (TabState)
- The VisibilityFilterSelector which depends on the filter value (VisibilityFilter)

Widgets changing the state are:

- The TabSelector which mutates the tab value dispatching a SetTabAction
- The VisibilityFilterSelector which mutates the filter value dispatching a SetVisibilityFilterAction
- Every TodoItem which mutates the list of todos dispatching a SetCompletedTodoAction

- The HomePage which starts the fetching process inside the *initState* method dispatching a LoadTodoAction

Note that the AppState does not contain the filtered list nor the stats. They are computed in the visualization layer respectively by the TodoView and Stats components. Moreover, the TodoView is rebuild every time a single todo changes because its viewmodel changes with it. This implies that the filtered list is recomputed every time a checkbox is tapped.

Understanding the Redux pattern was tough, but the implementation of the base functionalities was pretty linear. Most of the boilerplate came out from the action definitions and the StoreConnector widgets.

The optimization process leverages the fact that a StoreConnector rebuilds when its viewmodel changes.

These are the steps necessary to optimize the TodoView renderings:

- Make the TodoView viewmodel differs from the previous one only when a structural change occurs;
- Wrap every TodoItem into a StoreConnector to rebuild them independently from the TodoView.

Let's start with the first step. A TodoView widget should rebuild when a structural change occurs meaning that a todo is added or removed from the list or both cases together. Currently, the TodoView rebuilds at every state change because its viewmodel (a List<Todo>) always differs from the previous one due to how Dart compares objects. In fact, comparing two identical lists (containing the same values) evaluates to false because they are different instances. To change this behaviour, we can wrap the list into a new class and override its equality operator.

The new class is called _ViewModel and is private of the TodoView. Source code 5.3 reports its implementation.

Source Code 5.3: Comparing lists wrapping them in a viewmodel class

```
class _ViewModel {
  final List<Todo> todos;

  _ViewModel({required this.todos});
```

```
  @override
  bool operator ==(Object other) {
    if(other is _ViewModel) {
      List<int> ids = todos.map((todo) => todo.id).toList();
      List<int> otherIds = other.todos.map((todo) => todo.id).toList();

      return listEquals(ids,otherIds);
    }else{
      return false;
    }
  }
}
```

The equality operator simply maps both filtered lists to lists containing just the id before applying the *listEquals* function.

Let's proceed with the second step and wrap the TodoItem widget inside a StoreConnector. The converter function takes the store and selects the todo with the corresponding id. The id is passed to the TodoItem at its creation.

The equality operator for Todo instances checks recursively if all fields match and if the other object is of type Todo. This leads a TodoItem to rebuild when a field changes.

Here the summary of the collected data during the implementation process:

	lines of code	time	lines/time ratio	classes
base functionalities	539	9-11 h	0.81 l/m	6
rendering optimization	26	1 h	0.43 l/m	1

Table 5.2: Collected data during the implementation process based on Redux (Todos app)

5.2.4. Implementation based on BloC

This section describes the implementation and the architecture of the *todos* application (see Section 5.1) using a complete state management solution baed on BLoC. The solution uses: (a) the BLoC pattern to handle state, (b) context to dispatch it and (c) stream

components to kept UI synchronized. The implementation uses two Flutter packages: *bloc*(7.2.1), *flutter_bloc*(7.0.0) and *equatable*(2.0.0) (see Section 4.9).

I treated the entire application state as a shared state. In particular, the shared state contains:

- The list of todos,
- The list of filtered todos,
- The current visibility filter,
- The current tab value.

And it is divided in four blocs:

- The TodoBloc that handles the state of the list of todos,
- The StatsBloc that handles the state of the stats,(reacts to the TodoBloc stream of state)
- The FilteredTodoBloc that handles the state of the filtered list, (reacts to changes in the TodoBloc)
- The TabBloc that handles the state of the tab.

Class diagram in Figure 5.7 shows the application blocs with their corresponding states and events.

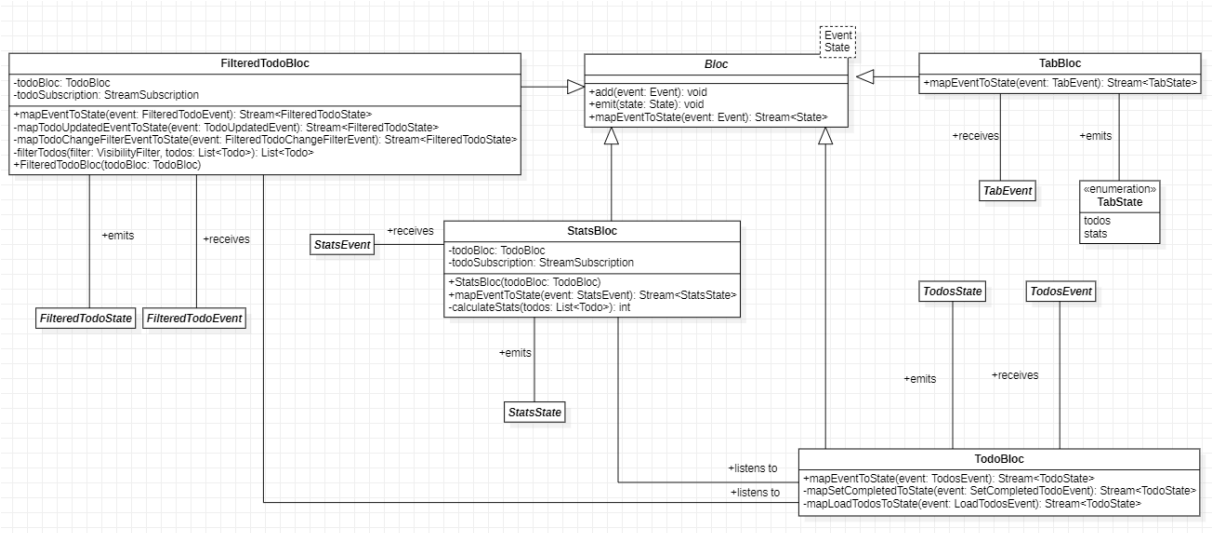


Figure 5.7: Blocs with their input events and output states used in the implementation based on the BLoC pattern

Important concepts in the diagram:

- The four blocs extend the Bloc class (from the *bloc* package) which provides the *add* method and the *emit* method. The former is used to receive events, the latter is used to emit states in the output stream;
- The Bloc class also requires the *mapEventToState* method to be overridden for each inheriting class;
- The FilteredTodoBloc and the StatsBloc listen to the TodoBloc using a StreamSubscription;
- The fetching of the list from the database is performed in the TodoBloc constructor at its instantiation;
- Every bloc receives abstract events and emits abstract states.

Class diagram in figure 5.8 shows the hierarchy of events and states. Events and states extend the abstract Equatable class that automatically overrides the equality operator, removing a great amount of boilerplate.

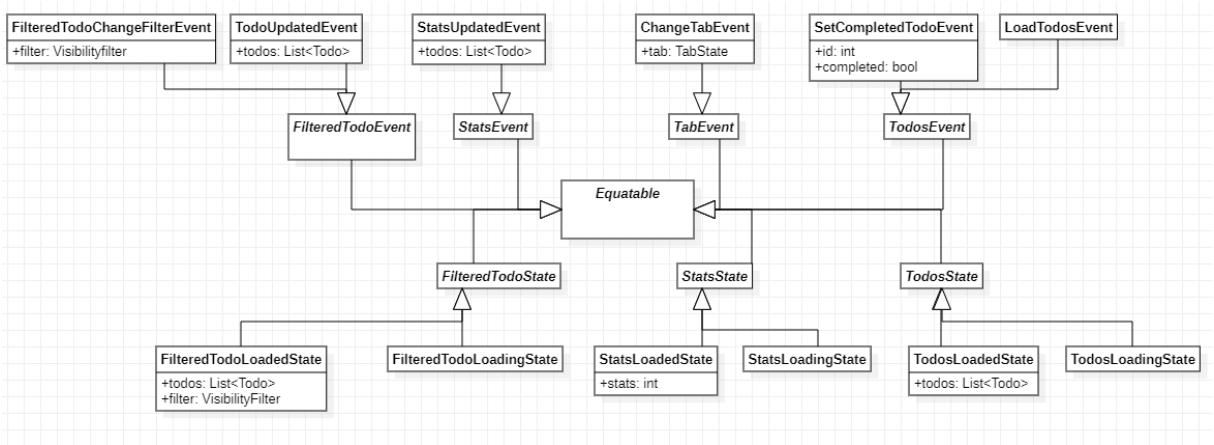


Figure 5.8: Hierarchy of events and states used in the implementation based on the BLoC pattern

Note: an application state in an application using the BLoC pattern is a collection of objects, each containing a specific part of the state, whereas an application state in an application using the Redux pattern is a single object of type AppState.

The application uses a single MultiBlocProvider at its root and multiple BlocBuilder widgets. Here a list of the widgets accessing the shared state with their corresponding bloc:

- The HomePage which listens to the TabBloc
- The TodoView which listens to the FilteredTodoBloc
- The Stat which listens to the StatsBloc
- The TabSelector which listens to the TabBloc
- The VisibilityFilterSelector which listens to the FilteredTodoBloc

Widgets changing the state are:

- The TabSelector which adds events in the TabBloc
- The VisibilityFilterSelector which adds events in the FilteredTodoBloc
- Every TodoItem which adds events in the TodoBloc

I would propose a sequence diagram to visualize the process of changing the completion of a todo. Figure 5.9 describes the interactions between the logic layer and a generic BlocBuilder listening to the FilteredTodoBloc outgoing states.

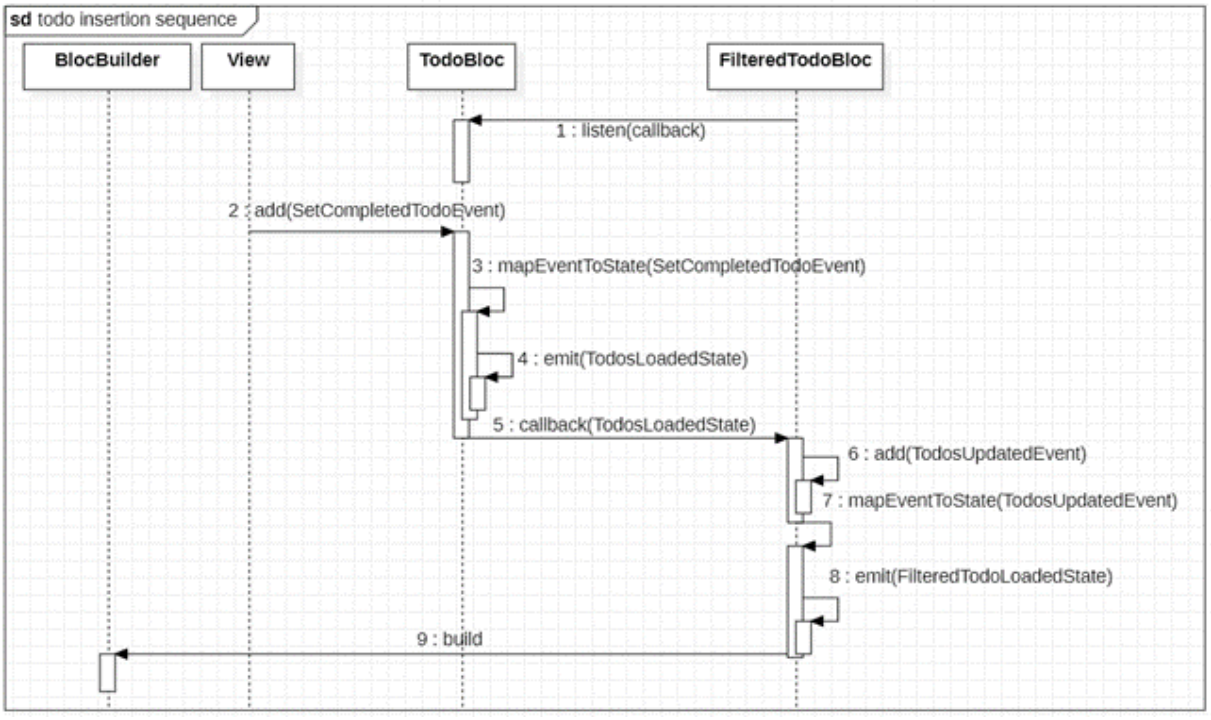


Figure 5.9: Insertion of a todo in the list with consequent view update

1. The FilteredTodoBloc listens to the stream of states of the TodoBloc providing a callback function to be called when a new state pops out;

2. A generic widget in the View adds a SetCompletedTodoEvent in the TodoBloc ingoing stream (for example a CheckBox tapped in a TodoItem);
3. The TodoBloc consumes the event through the *mapEventToState* method;
4. During the *mapEventToState* execution, a new state of type TodosLoadedState is emitted in the outgoing stream (The new state is visible by all the listener of the stream);
5. The FilteredTodoBloc reacts to the new state emitted by the TodoBloc calling the callback function of step 1;
6. During the callback execution, a new event of type TodosUpdatedEvent is added to the FilteredTodoBloc event stream;
7. The FilteredTodoBloc consumes the event calling the *mapEventToState* method;
8. During the *mapEventToState* execution, a new state of type FilteredTodosLoaded-State is emitted in the outgoing stream;
9. The BlocBuilder listening to the FilteredTodosBloc rebuilds due to the emission of the new state.

Note that the callback function used to react to changes in the TodoBloc does not emits any new state, **it adds a new event in the FilteredTodosBloc bloc ingoing stream**. The emission of the new state (if any) is left to the *mapEventToState* method. This workflow has two advantages:

- The entire logic that maps events to states is contained in a single method making much easier to understand the cause of a strange transition
- The emission of a new state is always preceded by an event receipt.

Theoretically, the callback function could incorporate its own logic and produce a new state without emitting any event, however, this would violate the action-based mutations policy of the BLoC pattern.

The optimization uses a specific field of the BlocBuilder widget called *buildWhen*. It takes a function that compares the current state and the previous one and returns a Boolean value on which basis the BlocBuilder rebuilds.

These are the steps necessary to optimize the TodoView renderings:

- Provide the TodoView *buildWhen* field with a function containing the rebuilding

logic

- Wrap every TodoItem into a BlocBuilder, this way they can rebuild independently by the TodoView

Overall, the procedure is similar to the one used with Redux (see Subsection 5.2.3) but presents an additional issue. With Redux, the application state is "fixed"; the structure is always the same and the contained data changes over time. For example, an AppState always contains a list, a visibility filter and a tab and the view is built based on the values each variable takes. When the application starts, the AppState contains an empty list meaning that the list of todos has not been fetched yet.

With BLoC, the application state is composed of a series of objects, each representing a specific aspect of the state. The structure of the application state changes over time with the data it contains. For example, the part of the state concerning the list can be represented by an object of type TodosLoadedState or an object of type TodosLoadingState. When the application starts, the state is of type TodosLoadingState and **does not contain any list at all.** This means that, with BLoC, it is not enough to check if the list changes over time but also that it actually exists.

s

Source Code 5.4: *buildWhen* field implementation in the TodoView BlocBuilder widget

```
buildWhen: (previous, next) {
  //if one of the states is of type Loading => rebuild
  return !((previous is FilteredTodoLoadedState) &&
      (next is FilteredTodoLoadedState) &&
  //else check for structural changes
      checkStructuralChange(previous.todos, next.todos));
}
```

Source Code 5.5: *buildWhen* field implementation in the TodoItem BlocBuilder widget

```
buildWhen: (previous, next) {
//if one of the states is of type Loading => rebuild
  return (next is TodosLoadedState &&
      previous is TodosLoadedState &&
//else check for changes in the todo
```

```
      todoIsChanged(previous.todos, next.todos, id));
}
```

Table 5.3 shows a summary of the collected data during the implementation process.

	lines of code	time	lines/time ratio	classes
base functionalities	744	10-12 h	1.03 l/m	24
rendering optimization	28	6-8 h	0.066 l/m	0

Table 5.3: Collected data during the implementation process based BLoC (Todos app)

5.2.5. Implementation based on MobX

This section describes the implementation and the architecture of the todo application (see Section 5.1) using a complete state management solution. The solution uses: (a) the MobX library to handle the application state and (b) to keep it synchronized with the UI, (c) context to dispatch it. The implementation uses five Flutter packages: *mobx* (2.0.5), *flutter_mobx* (2.0.2),*provider* (6.0.1), *build_runner*(2.1.4) and *mobx_codegen*(2.0.4) (see Section 4.10).

I treated the entire application state as a shared state except for the tab state. In particular, the shared state contains:

- The list of todos,
- The list of filtered todos,
- The current visibility filter.

I treated the current tab value as an ephemeral state. It is contained in the HomePage and is handled with the *mobx* package using an Observable variable.

Class diagram 5.10 shows the abstract class TodoStore used to generate the code in the *todo_store.g.dart* file.

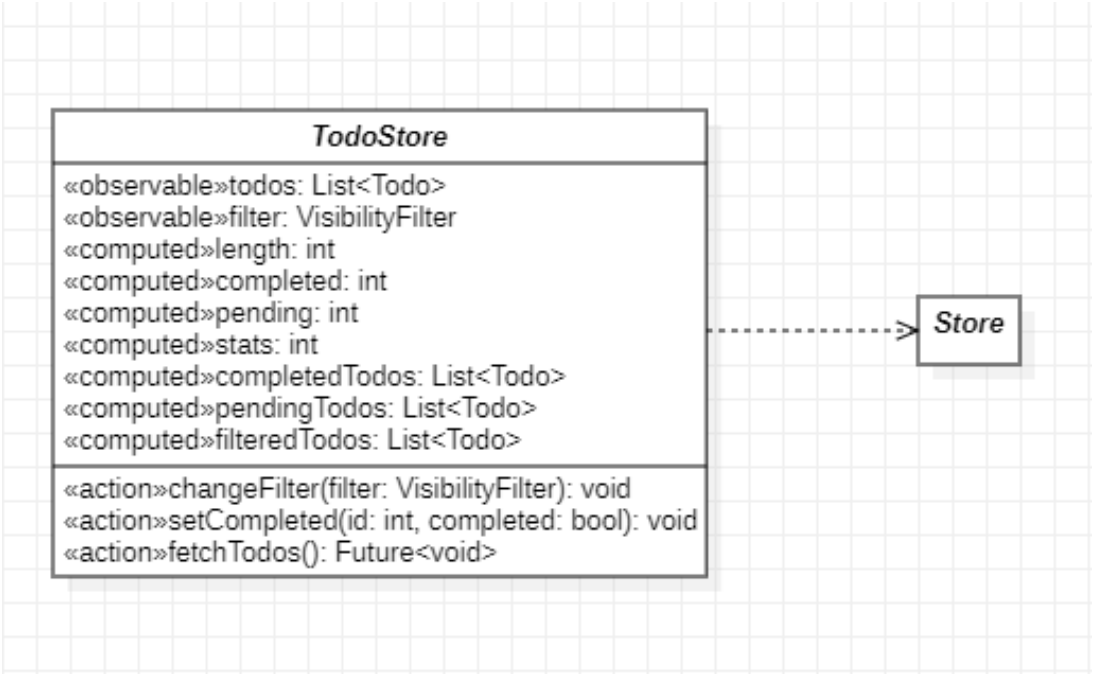


Figure 5.10: TodoStore class diagram of the implementation based on MobX

The application uses a Provider widget, situated at the root, to dispatch the TodoStore to the sub-tree. Several Observer widgets react to changes in the Store. Here a list of the position of the Observer widgets with their observed variable:

- the HomePage that observes the tab variable (local to the HomePage)
- the Stats that observes the stats variable
- the TodoView that observes the filteredTodos variable
- the VisibilityFilterSelector that observes the filter variable

Here the list of the widgets that update the Store:

- The Provider widget that calls the *fetchTodos* action at its creation
- The VisibilityFilterSelector that calls the *changeFilter* action
- The TodoItem that calls the *setCompleted* action

The optimization process was fast and easy as anticipated/foreseen in Section 3.10. It required two steps:

- Make the Todo model observable;
- Wrap every TodoItem in an Observer widget.

The first step basically requires to mark every variable in the Todo model as observable and to generate the corresponding code. After step 2, the MobX reactivity system is able to distinguish changes in the list and changes in a single todo.

This was the simplest optimization process and did not require any logic definition. Even if the other optimization processes were not difficult, they introduced another potential source of errors and performance issues that could be automatized and/or removed, as the MobX package does.

I would like to point out that implementation with MobX is the only one that required to modify the model definitions (in the optimization part). As stated in Section 3.10, the usage of the MobX package makes the application even more dependent from the external packages.

Table 5.4 shows a summary of the collected data during the implementation process.

	lines of code	time	lines/time ratio	classes
base functionalities	441 (+110)	5-6 h	1.22 l/m	2 (+1)
rendering optimization	15 (+ 47)	30 m	0.5 l/m	1 (+1)

Table 5.4: Collected data during the implementation process based on MobX (Todos app)

6 | The Biometrics app

This Chapter describes another implementation process regarding a more complex application with respect to the one in Chapter 5.

6.1. The objective

The target of this development process is to generate an application with the following characteristics:

- A large shared state,
- Multiple pages (10+),
- Several async tasks,
- An elevated number of lines of code(1500+).

The resulting application collects several data regarding the user biometrics and activities and combines them to produce statistics about the user health and goals. In particular, the application connects to a mocked external device (such as a smartwatch) which asynchronously produces data concerning the user biometrics (heart rate, temperature etc). The application provides a way of manually inserting information about the user daily activities (such as the eaten food and the performed activities) and to set arbitrary daily goals (daily food income, number of steps etc). Data collected through the device are merged with data inserted by the user to compute and visualize a recap of the user current situation.

To achieve my goal, I started with a list of requirements the application must satisfy:

1. Handle authentication
    - Provide a login/register page
    - Change the application behaviour based on the current authentication state
2. Allow theme switching (dark, light)

3. Handle connection to an external device
    - Provide a procedure, accessible only if authenticated, to search, select and connect to a fake device,
    - Provide a mechanism to save the received data in a buffer and to periodically store them to a fake database when the buffer is full
    - Provide a page where to visualize the last received data
4. Handle a user daily diary
    - Search, select and insert a food in a daily diary
    - Search, select and insert a physical activity in a daily diary
5. Associate the user with a set of preferences
6. Visualize the user current situation computed merging the device data, the daily diary and the user preferences

6.2. The implementation

I will not enter in the details of each implementation process, instead I will propose just the relevant aspects of each architecture.

Let's start visualizing the shared parts between different implementations using the class diagram in Figure 6.1. It shows the models used to implement the application core.

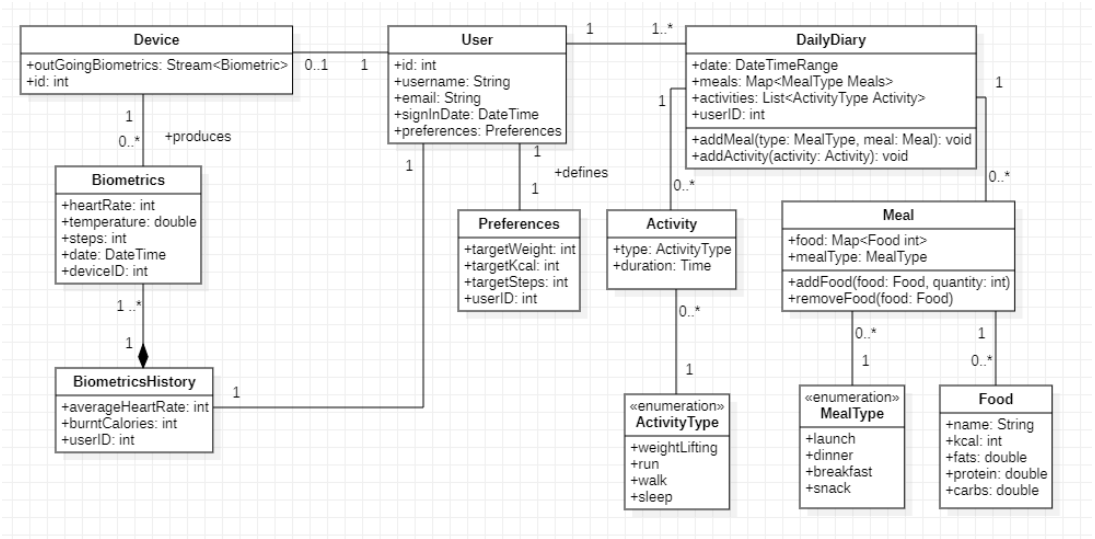


Figure 6.1: Class diagram about the models shared between different implementations (Biometrics app)

- Each user is associated with a single Preferences object and each Preferences object belong to a single user
- Each user is associated with a single BiometricsHistory composed of several Biometrics
- A user can be associated with zero or one device. Some features are available only if the device is connected.
- A user is associated with several DailyDiarys

The application shared state is composed of:

- The current user authentication
- The current state of the device
- The current user biometrics
- The user biometrics history
- The daily diary
- The user preferences

Each part of the shared state has a component (or a page) where to be visualized and modified. Diagram 6.2 show the dependencies between substates.

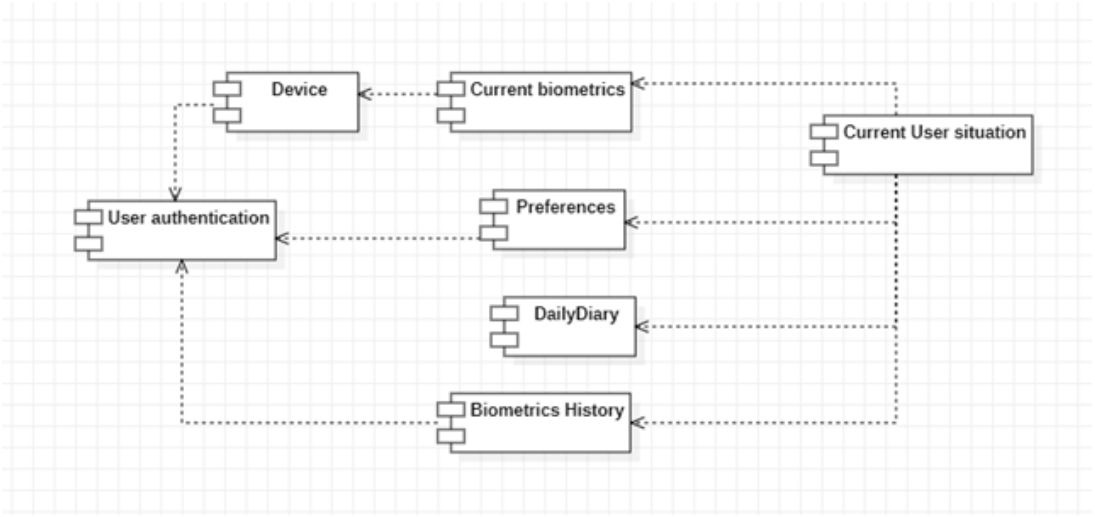


Figure 6.2: Dependencies between features (Biometrics app)

- Authentication enables multiple other features. Only daily diary features are available to not authenticated users
- The device connection procedure is available only after the authentication process
- Once a device has been set, it is possible to visualize the current user biometrics in a dedicated page
- After the authentication process it is possible to change the user preferences in the homepage
- Biometrics history is available after the authentication process but does not require a connected device
- It is possible to visualize the current situation in a dedicated page, available to authenticated users with a connected device only.

The application is composed of multiple pages that depends on one or more aspects of the shared state.

6.3. The external device

To make the communication with the external device more realistic I decided to use a stream. I wanted the application to receive biometrics asynchronously and autonomously instead of fetching them from the device.

The device contains a biometrics generator that forwards randomly generated biometrics in a stream at random intervals. During the device set up, the application subscribes to new biometrics coming out of the stream caching them and storing them to a fake database when they reach an arbitrary number (15 by default).

The first implementation is the one based on plain setState. Basically, the entire shared state is contained in a state object in the Homepage. The state object contains several private state changing functions that are forwarded and used by the other pages to mutate the shared state.

The procedure used to cache and store the received biometrics is handled using a StreamSubscription in the Homepage. It stores the received biometrics in a temporary list until it reaches 15 elements. At that point, the list of biometrics is saved in the fake database and flushed.

The implementation based on BLoC uses two separated blocs to handle the shared state regarding biometrics:

- One is called CurrentBiometricsBloc and contains the last received biometrics
- The other is called BiometricHistoryBloc and reacts to state changes in the CurrentBiometricsBloc bloc to perform biometrics caching and storing.

Figure 6.3 shows the architecture of the shared application state concerning the biometrics.

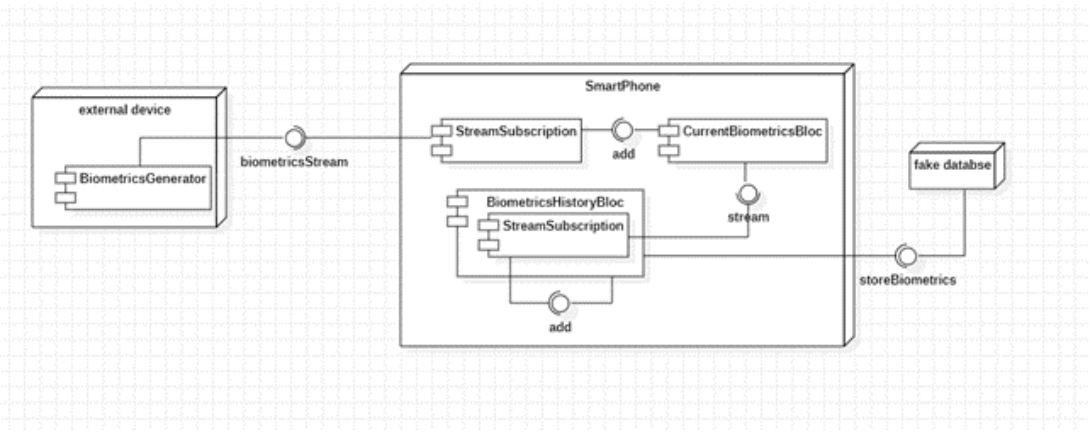


Figure 6.3: Architecture of the application based on the BLoC pattern (Biometrics app)

- The application that runs on the smartphone sets up a StreamSubscription during the device connection procedure. The subscription reacts to biometrics coming

out of the device stream and generates an event of type ReceivedBiometrics in the CurrentBiometricsBloc,

- The CurrentBiometricsBloc emits a new state of type CurrentBiometrics,
- The BiometricsHistoryBloc creates a StreamSubscription that listens to states coming out of the CurrentBiometricsBloc stream and emits an event of type ReceivedBiometrics in the BiometricsHistoryBloc,
- The BiometricsHistoryBloc adds the just received biometrics in a list and in case the list has more than 15 elements it stores them in the database and flushes the list.

The application based on Redux contains two variables in the appState: one indicating the current biometrics and the other accumulating the received biometrics.

Figure 6.4 shows the architecture of the application concerning the biometrics shared state.

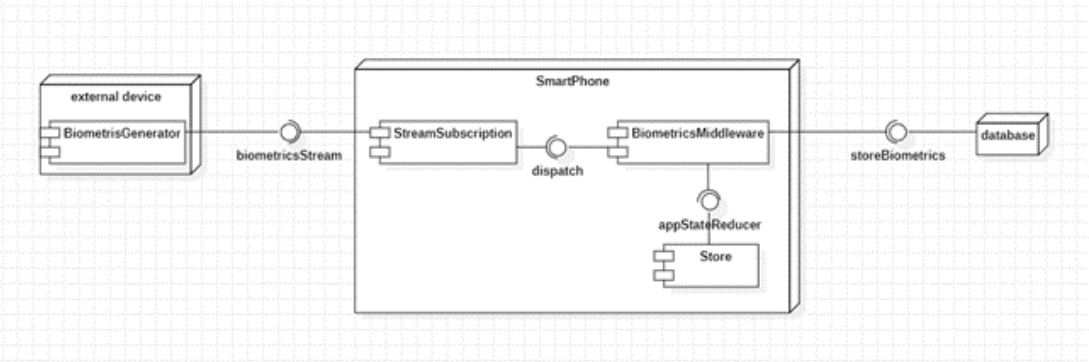


Figure 6.4: Architecture of the application based on the Redux pattern (Biometrics app)

- The application that runs on the smartphone sets up a StreamSubscription during the device connection procedure. The subscription reacts to biometrics coming out of the device stream dispatching an action of type ReceivedBiometrics.
- The BiometricsMiddleware reacts to the newly received action in two ways depending on the current appState state. In case the buffer is not full it just forward the action to the appStateReducer. In the other case, before forwarding the action it flushes the buffer and stores it to the database.

The application based on MobX contains two variables in the Store: one indicating the current biometrics and the other accumulating the received biometrics.

Figure 6.5 shows the architecture of the application concerning the biometrics shared state.

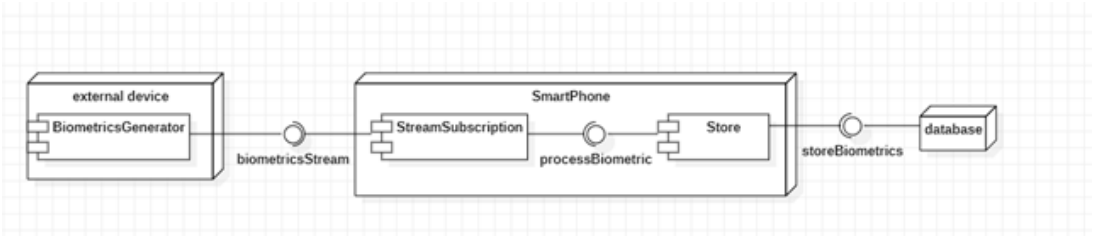


Figure 6.5: Architecture of the application based on MobX (Biometrics app)

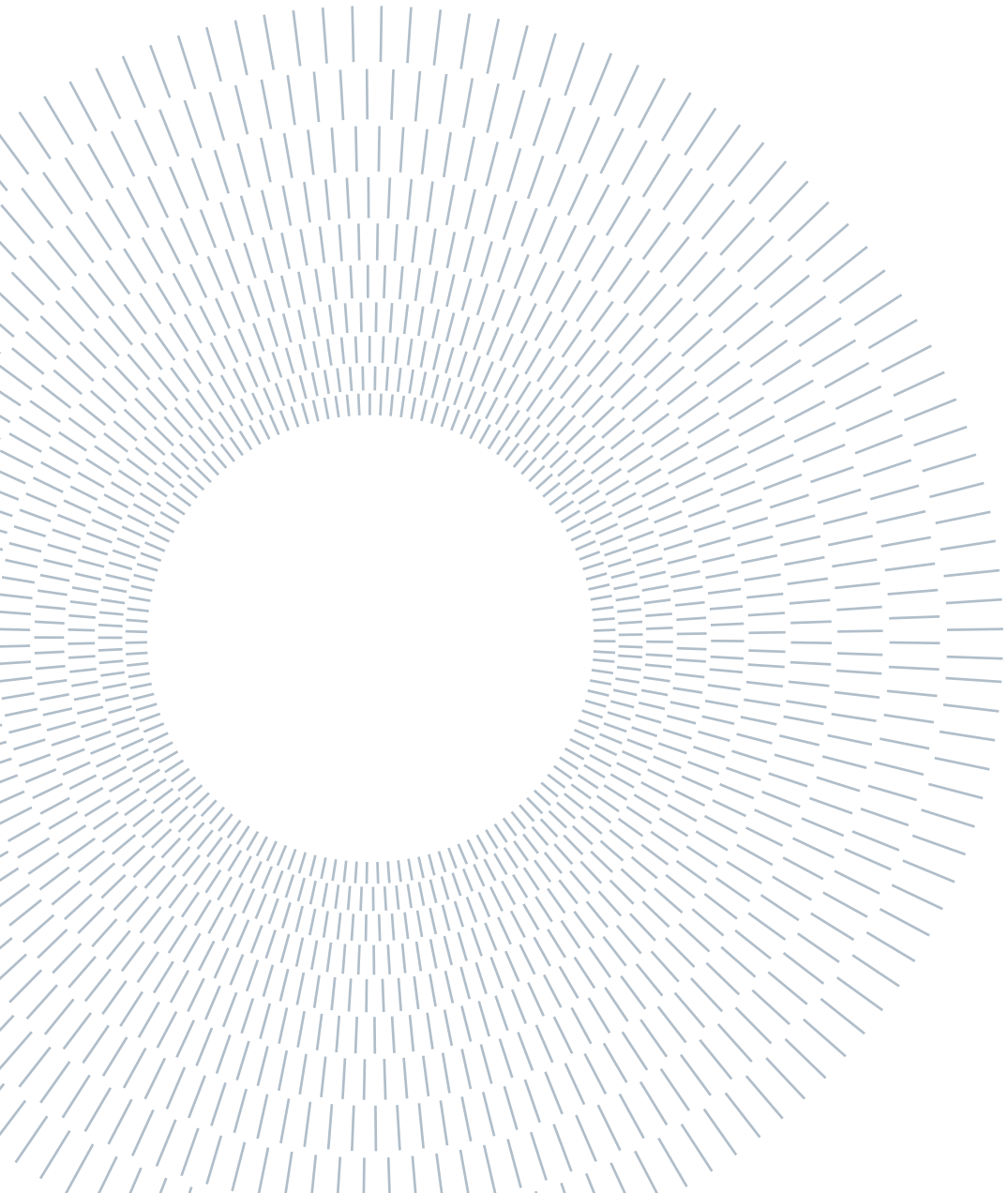
- The application that runs on the smartphone sets up a StreamSubscription during the device connection procedure. The subscription reacts to biometrics coming out of the device stream calling the *processBiometric* method of the Store with the received biometrics.
- The *processBiometrics* method sets up the current biometrics value to the newly received one and checks is the buffer is full. In case it is, it stores the buffer in the database and flushes it.

Table 6.1 reports the data collected during the implementation process.

solution	lines of code
setState	1538
InheritedWidgets	1513
BLoC	1889
Redux	1717
MobX	1481

Table 6.1: Collected data during the implementation process about the lines of code required by different state management solutions (Biometrics app)

Redux and BLoC still require more lines of code with respect to setState but MobX actually requires fewer.



7 | The Pixels app

7.1. Pixels experiment

This chapter takes a cue from an article I found on the web [8]. The article proposes an ad hoc created application where the synchronization process used by Redux performs worse than the one by MobX (in terms of application performances). I replicated the experiment in the Flutter framework introducing to the experiment a solution using BLoC, a solution using setState and a solution using InheritedWidget but I was not able to find out the same differences in perfomance. However, I highlighted some other intresting facts about the synchronization process.

7.1.1. The objective

The objective of this experiment is to test the impact of the synchronization process on the application performances. The experiment compares the performance of an application using an externalized state and an application using state objects. To highlight the differences in performance, the experiment puts in place a sort of "stress test" of the synchronization process. In particular, the experiment considers and application with:

- An elevated number of components,
- frequent state changes,
- changes that affect all components at once.

Since, at every state change, the information in the externalized state must be propagated to the visualization layer, I expect the application using an externalized state to require a higher CPU usage with respect to the one using plain state objects when performing the same series of tasks.

Here the list of the tested synchronization processes:

- Redux+ Stream components

- MobX (observer components)
- BLoC + Stream components

7.1.2. The application

The application used in this experiment is composed of a single page with two PixelContainers. A PixelContainer is composed of 64^2 Pixels.

A Pixel is just a Container widget with a height, a width and a colour representing the current activation of the pixel. (White: inactive, black: active)

The state of the pixels is shared between the two PixelsContainers, otherwise would be meaningless to use a state management solution at all.[8]

The state of the pixels is handled using a list of lists of Boolean values. Here the characteristics of the tested synchronization processes:

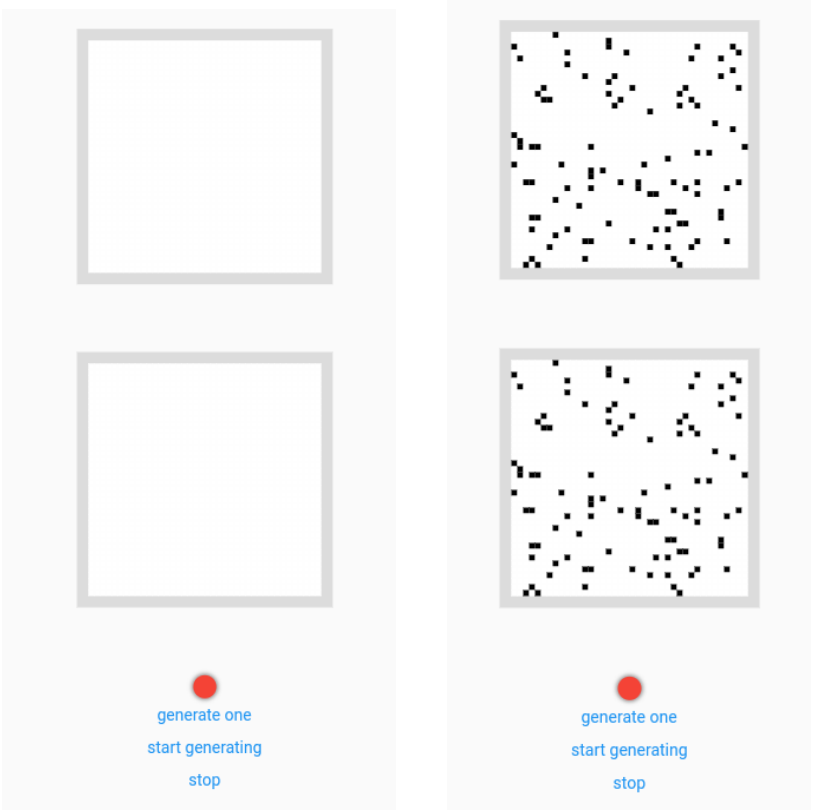
- Redux
  The state resides in a single store and the view is provided with a connector (StoreConnector) for each pixel. Since every connector creates a stream, there are $64^2 * 2$ active streams during the entire execution.
  When the state changes, each connector receives the new state from the store, computes its viewmodel and rebuilds.
- MobX
  The state resides in a store composed of an observable list of lists of Booleans. The view is provided with a connector (Observer) for each pixel. The Store is built in a way that forces the MobX reactivity system to rebuild every connector at each state change.
- BLoC
  The state resides in a single bloc and the view is provided with a connector (BlocBuilder) for each pixel. Since every connector creates a stream, there are $64^2 * 2$ active streams during the entire execution.
- setState and InheritedWidget
  in this case the state resides in state objects, there is no need for a synchronization process.

7.1.3. The experiment

Is very crucial for the experiment that the number of rebuilds at each state change is equal for every tested synchronization process. This allows to consider the time required to rebuild all the widgets equal, and to take the difference in performances between applications with an externalized state and the application using state objects as the overflow introduced by the synchronization process.

The experiment consists in filling the PixelContainers with 64^2 pixels each and changing their state sequentially for 100 times. A new state change starts only when the previous one has been processed and synchronized with the UI.

Figure 7.1 shows the UI of the experiment page before and after the generation process.



(a) start of the experiment (b) end of the experiment

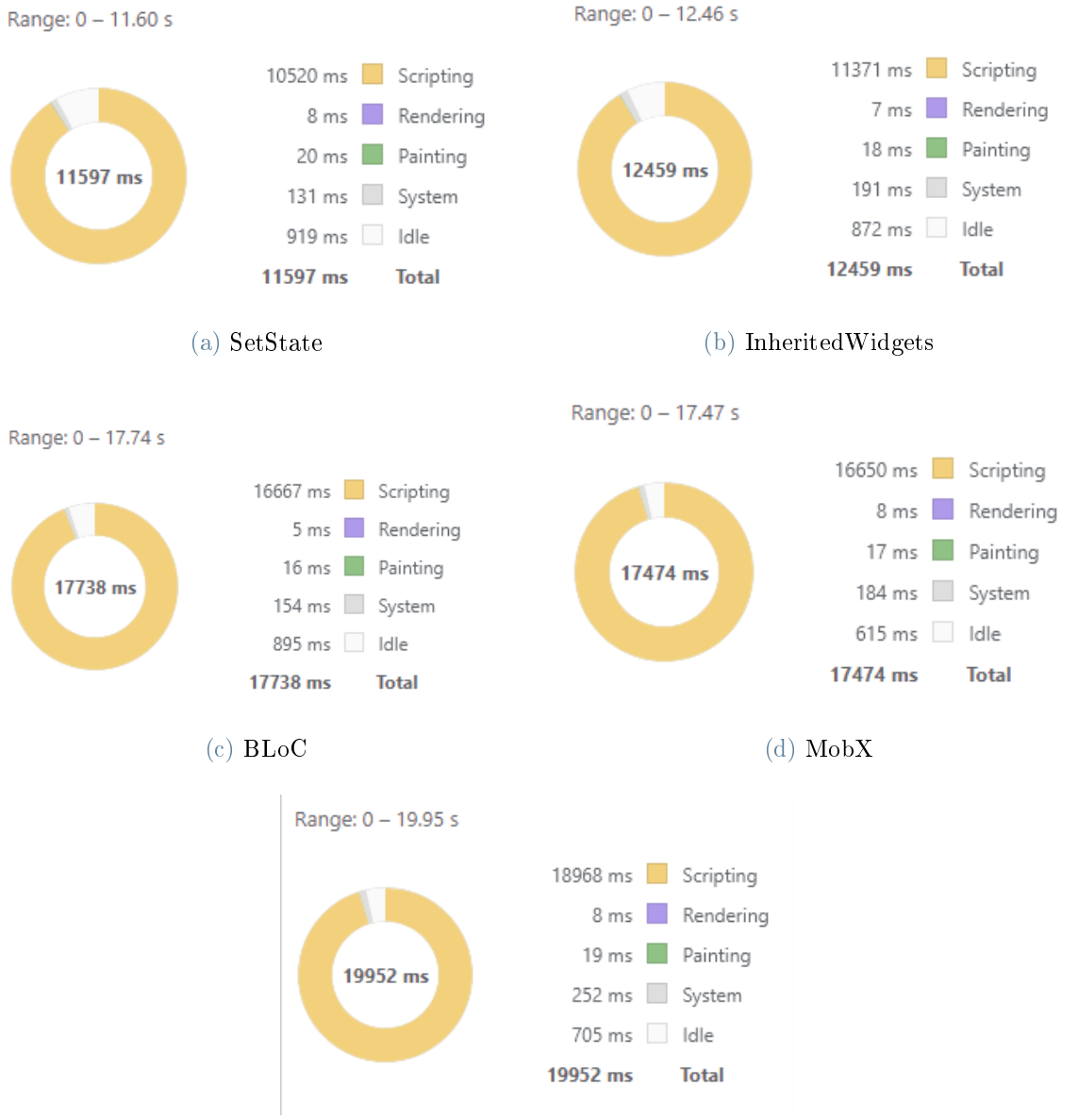
Figure 7.1: UI for the experiment used to test the synchronization process.

7.1.4. Collected data

Measurements start when the first state change occurs and stop when the 100th state change is properly rendered by the UI. Measurements are taken from an execution of the application in profile mode on Chrome (The application run as a Web application) using the Chrome profiling tool. The Chrome profiling tool produces a pie chart that indicates:

- the total time passed from the first activation to the last one,
- the time spent in each activity (scripting, rendering, etc..)

The "Total" value indicates the overall execution time of the activation process. The greater "total" is the longer the application took to complete all the 100 state changes with their synchronization. The longer the application took the higher the effort was for the CPU to perform all the computations.



(a) SetState (b) InheritedWidgets

(c) BLoC (d) MobX

(e) Redux

Figure 7.2: Measurements regarding the activation process of 100 Pixels taken from the Chrome profiling tool.

Pie charts in Figure 7.2a and 7.2b are collected from an execution based on plain setState and InheritedWidgets and do not require any synchronization process because the application state already resides in a state object. Since they do not require to perform any synchronization process when the state changes, they have a lower amount of work to perform and consequently a "total" value that is lower with respect to the other executions.

Pie charts in Figures 7.2e, 7.2c and 7.2d are collected from executions based respec-

tively on Bloc, Redux and Mobx. They all have a higher "total" value with respect to the executions based on setState and InheritedWidget. The overflow is somehow generated by the synchronization process the state management solution must perform. Let's try to find out why:
Executions based on Redux and Bloc have an elevated number of active streams to which new states are forwarded at every state change. Moreover, stream components used in the application based on Redux (from *flutter_redux*) compute a view model of the application state at every state change, leading the CPU to perform additional computations and the pie chart to have a higher "total" time with respect to the others.

It is not clear what is happening under the hood in the application based on MobX. I think that the overflow generated by the synchronization process is caused by the subscriptions and un-subscriptions to the observables the MobX reactivity system must perform at every state change.

Overall, the experiment pointed out that the synchronization process of an externalized state with the UI increases the resources consumption (CPU usage) when the number of rebuilds is equal. In general, an application using an externalized state requires a higher CPU effort with respect to an application based on plain setState to perform the same number of widget updates.

In this particular case, the application performances have been aggravated by a 50-72%.

8 Conclusions

State management solutions provide several benefits in multiple aspects of the development process of complex mobile applications. The approach they use relies on separating the application state from the visualization layer and to mutate it through predefined actions. This process greatly increases the application predictability, testability and flexibility but also introduces two new problems.

The former is the necessity to define standard interfaces to access and communicate with the application state. These interfaces generate the, so called, boilerplate.

The latter is the necessity to synchronize the externalized state with the UI. Since Flutter requires information to reside in state objects to be properly visualized, additional tools are required to inject the application state in state objects efficiently, consistently, and automatically. The complexity of this process highly depends on the target framework, but, in general, consumes resources and generates even more boilerplate.

In this last chapter I will use the collected data throughout the thesis to quantify the boilerplate introduced by different state management solutions and the impact of the synchronization process on the application performances.

8.1. Boilerplate

Histogram 8.1 reports the data collected during the implementation processes and about the lines of code required to implement three different applications with incremental complexity. X axis reports the three development processes, whereas Y axis indicates the lines of code produced by each solution in the development process.

Each application has been implemented using five different state management solutions. Two of them, setState and InheritedWidgets, are provided by the Flutter framework and do not use any externalized state, the other three (BLoC MobX Redux) are provided by

external libraries.

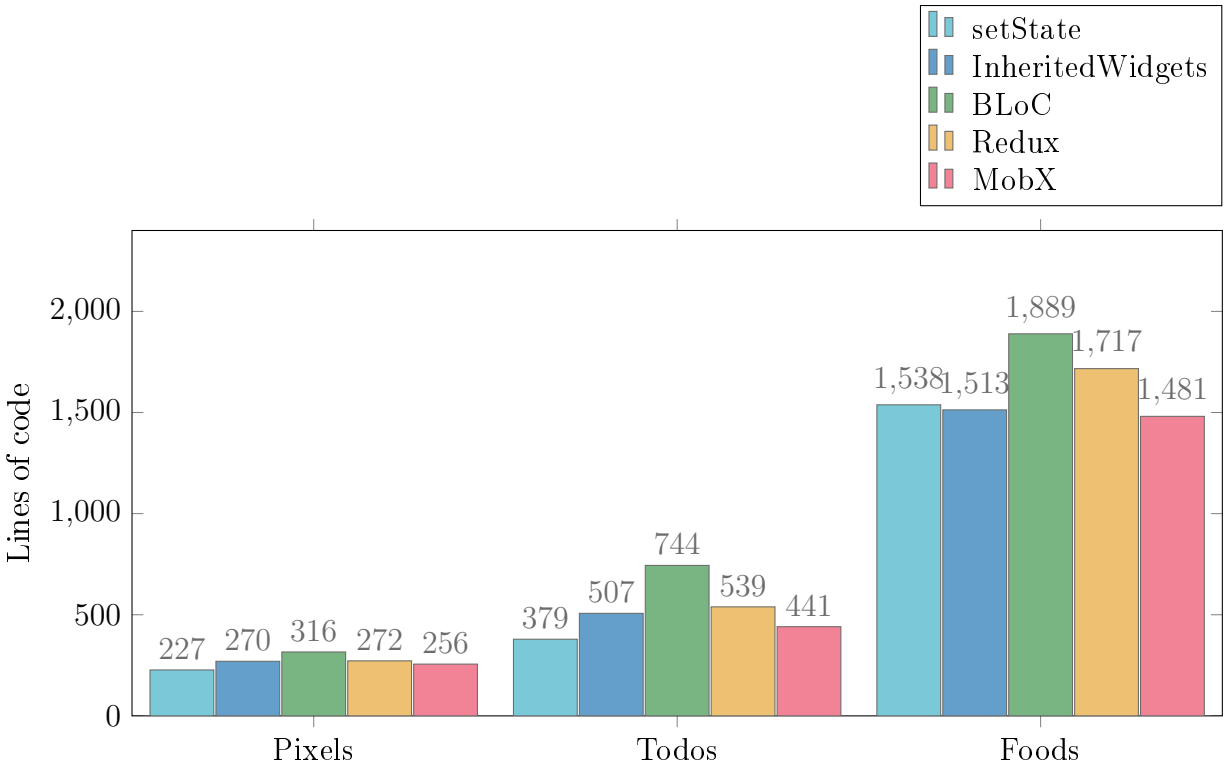


Figure 8.1: Histogram representing the lines of code required by each state management solution to implement three different applications with incremental complexity

Yellow bars are taken from the implementation process using setState (the basic framework features to handle the application state) and can be intended as the baseline number of lines of code required to implement the application. What emerged is:

- The BLoC pattern (using stream components) has the highest required boilerplate, it adds from a minimum of 23% to a maximum of 96% lines of code with respect to the ones used with plain setState.
- The Redux pattern (using stream components) adds from a minimum of 11% to a maximum of 42% lines of code with respect to the ones used with plain setState.
- MobX has the lowest required boilerplate. It adds to the two applications with lowest complexity about 12-15% lines of code with respect to to the ones used with plain setState. In the most complex application, MobX requires fewer lines of code with respect to the ones used with plain setState. It is important to point out that, most of the boilerplate introduced by MobX is hidden by the code generator (not considered in the histogram) that hides and adjunctive 20-25% of boilerplate.

Diagram in Figure 8.2 reports the lines of code added by each state management solution based on the one required with plain setState. In particular:

- on the X axis the lines of code required to implement an application with plain setState,
- on the Y axis the percentage difference of the lines of code required to implement an application using setState the state management solution.

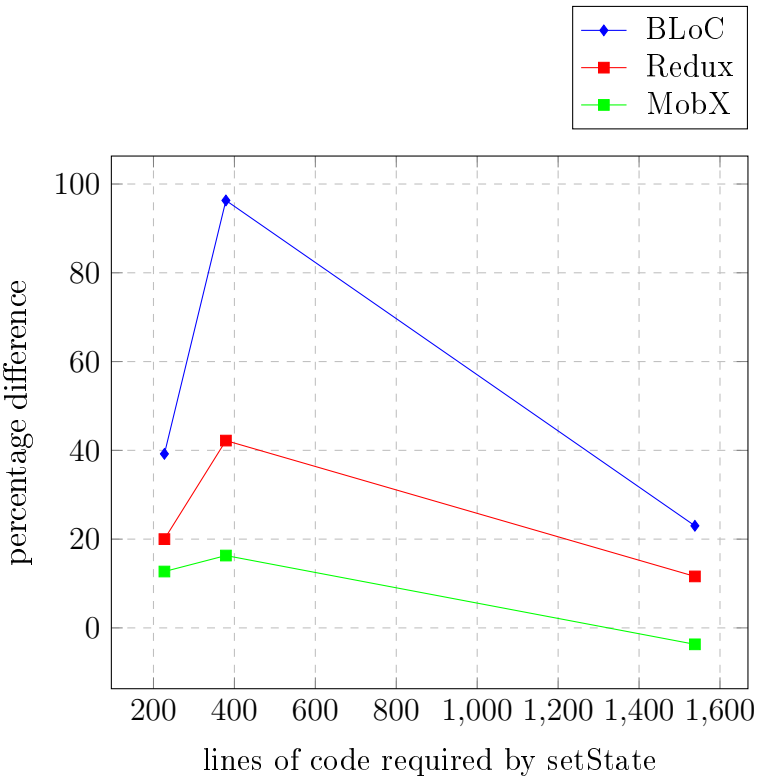


Figure 8.2: Trend of the percentage lines of code added by state managements sollutions with respect to plain setState

My hipothesis is that the amount of boilerplate required by Redux , BLoC and MobX gets amortized by the application size and complexity growth. When complexity reaches a certain threshold, the lines of code required to perform a modification using plain setState are higher or equal to the ones required by Redux, BLoC and MobX.

Both curves in diagram in Figure 8.2 have a starting pick. This pick derives from the necessity to set up the solution general interface. For BLoC this concerns the definition of the blocs with their states, events, the mapping function and the BlocProvider. For Redux this concerns the definition of the appState model, the reducer and the Store-

Provider. Moreover, in the starting fase of the application development, the boilerplate introduced by the *props drilling problem* in the implementation based on setState is neglectable (small widget tree). As the application grows, the lines of code required by setState to accomplish simple tasks grows fast (I would say quadratically), whereas the lines required by Bloc and Redux grows linearly. Adding new features to a bloc or to a store requires a lower effort with respect to setState, also because solutions like BLoC and Redux avoid code duplication, setState usually does not.

About MobX, its trend is similar to the BLoC and Redux ones but has a less accentuated pick at the beginning. This behaviour is explained by the fact that the code generator automatically creates the interface of the solution, doing so it mitigates the required initial boilerplate.

Since the data collection process considers only three applications, each of them with a limited number of lines of code , my hypothesis has not a great support. More collected data concerning an application with an higher amount of lines of code should be provided.

8.2. Synchronization process

About the synchronization process, the experiment in Chapter 7 pointed out that it requires additional computational effort to be performed. Histogram in Figure 8.3 reports the total execution times of 100 consecutive state changes taken from five applications based on different state management solutions, each of them rebuilding the same number of widgets at every state change.

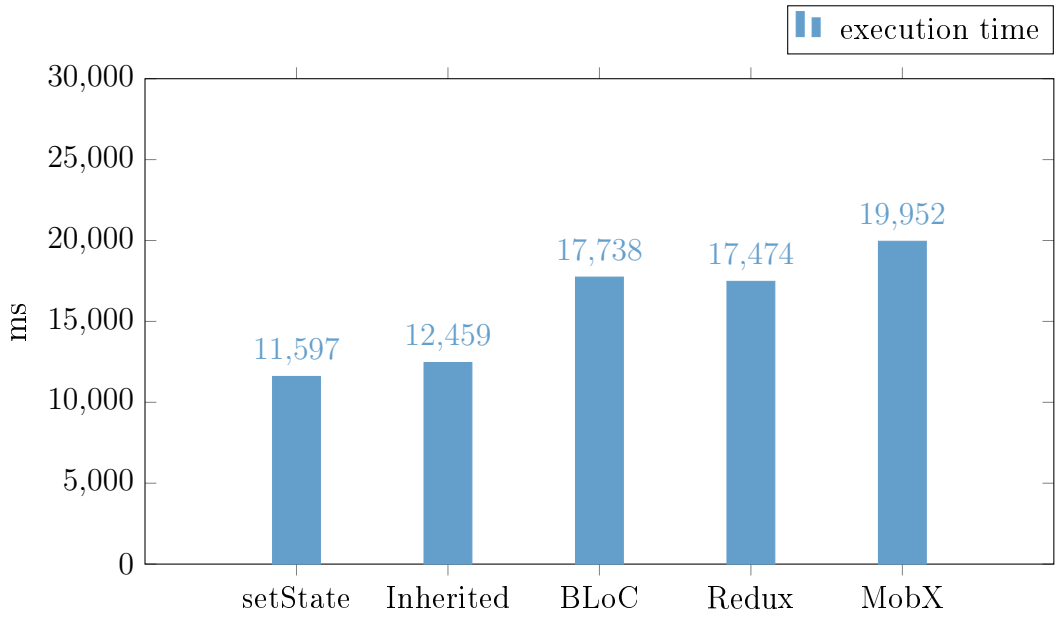


Figure 8.3: Histogram representing the time required by different state management solution to complete the same number of state changes (Pixels app)

Applications using setState and InheritedWidgets do not require any synchronization process as the application state already resides in state objects. It is clear by Figure 8.3 that the synchronization process required by the applications based on an externalized state (Mobx, Redux, BLoc) adds from 6 to 8 seconds to the overall execution time of the experiment. Extrapolating the percentage difference between the execution time of the application based on setState and the execution times based on an externalized state emerges that the synchronization process adds from 50 to 72% computational effort.

This is caused by the necessity to handle a great number of streams and viewmodels computation for the applications based on BLoC and Redux and by the MobX reactivity system in the application based on MobX.

Can the additional computational effort significantly decrease the application FPS? In my opinion no. The answer actually depends on the context but, in general, the experiment is clearly biased to stress out the synchronization process since the number of connectors and the ratio between connectors and rebuilds is too high. In a realistic scenario, the number of connectors is much lower. An application would never contain 8196 connectors that simultaneously change state and, usually, a single connector rebuilds more than one widget.

We can say that, in a real application, the synchronization process will add an over-

flow lower or equal to the 72% of the computational effort required by setState to carry on the same number of rebuilds.

8.3. Future work

This work can be enlarged in multiple directions:

- Introducing other state management patterns or libraries to the comparison,
- Producing one or more adjunctive applications with a higher number of lines of code and a greater complexity,
- Adding one or more dimensions to the comparisons.

Moreover, this work can be taken as a starting point to define a sort of recurring model for application states (mostly the shared parts). In fact, an application state usually contains recurring patterns that can be extrapolated and given as input to a process that automatically generates the required boilerplate. Clearly, this process is going to be more effective on solutions like Redux and BLoC that requires the highest amount of boilerplate. Some examples of recurring patterns I found out during this thesis are:

- Handling authentication
  This is a clear case where it is possible to define a sort of model of the process to be used to automatically generate the code.
- Handling multiple views and/or operations on a collection of items
  Another common recurring pattern is to deal with the standard operations for collections of objects, such as: inserting, deleting and mutating elementes. Moreover, like in the case of the filtered list, is also usually required to filter the collection by an arbitrary property.

Bibliography

[1] AlDanial. Count lines of code (cloc), 2021. URL https://github.com/AlDanial/cloc.

[2] D. Boelens. Reactive programming - streams - bloc, 2018. URL https://www.didierboelens.com/2018/08/reactive-programming-streams-bloc/.

[3] J. Gohil. Understanding state management in front-end paradigm, 2020. URL https://www.linkedin.com/pulse/understanding-state-management-front-end-paradigm-jitendrasinh-gohil.

[4] J. T. Lapp. Understanding provider in diagrams — part 1: Providing values, 2019. URL https://medium.com/flutter-community/understanding-provider-in-diagrams-part-1-providing-values-4379aa1e7fd5.

[5] J. T. Lapp. Understanding provider in diagrams — part 2: Basic providers, 2019. URL https://medium.com/flutter-community/understanding-provider-in-diagrams-part-2-basic-providers-1a80fb74d4e7.

[6] J. T. Lapp. Understanding provider in diagrams — part 3: Architecture, 2019. URL https://medium.com/flutter-community/understanding-provider-in-diagrams-part-3-architecture-a145e4fbbde1.

[7] C. Panel. React – working with data, 2022. URL https://www.codingpanel.com/lesson/react-working-with-data/.

[8] T. Pangsakulyanont. An artificial example where mobx really shines and redux is not really suited for it, 2016. URL https://hackernoon.com/an-artificial-example-where-mobx-really-shines-and-redux-is-not-really-suited-.42tpw3uw9.

[9] F. Team. Inheritedwidget class, 2022. URL https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html.

[10] F. Team. Inheritedmodel<t> class, 2022. URL https://api.flutter.dev/flutter/widgets/InheritedModel-class.html.

[11] F. Team. Reactive user interfaces, 2022. URL https://docs.flutter.dev/resources/architectural-overview#reactive-user-interfaces.

[12] F. Team. Differentiate between ephemeral state and app state, 2022. URL https://docs.flutter.dev/development/data-and-backend/state-mgmt/ephemeral-vs-app#there-is-no-clear-cut-rule.

[13] R. Team. Redux application data flow, 2022. URL https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow#redux-application-data-flow.

[14] R. Team. Three principles, 2022. URL https://redux.js.org/understanding/thinking-in-redux/three-principles.

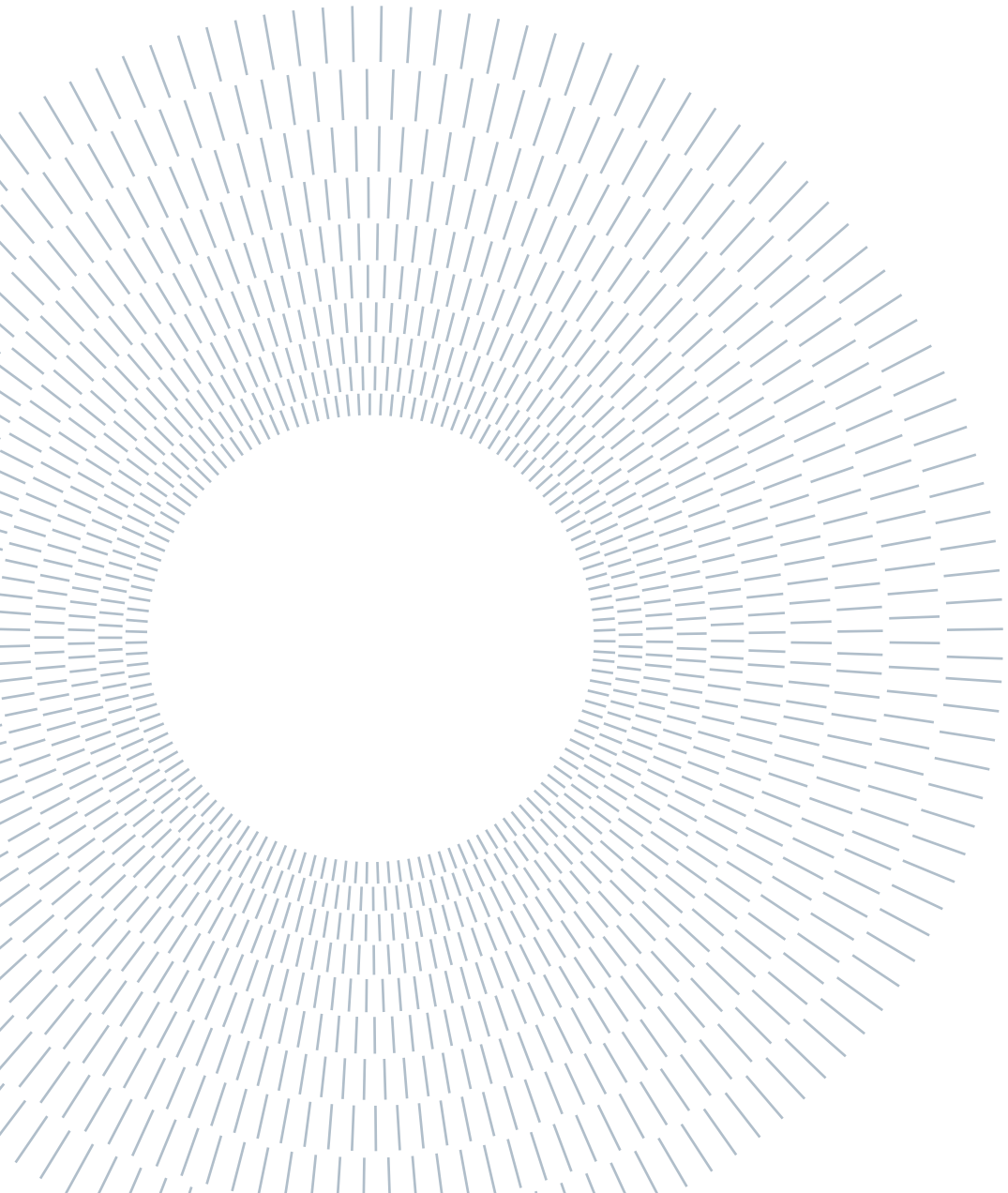
[15] M. Weststrate. Becoming fully reactive: an in-depth explanation of mobx, 2015. URL https://hackernoon.com/becoming-fully-reactive-an-in-depth-explanation-of-mobservable-55995262a254.

[16] M. Weststrate. What does the t(transparent) in tfrp mean?, 2016. URL https://github.com/mobxjs/mobx/issues/220.

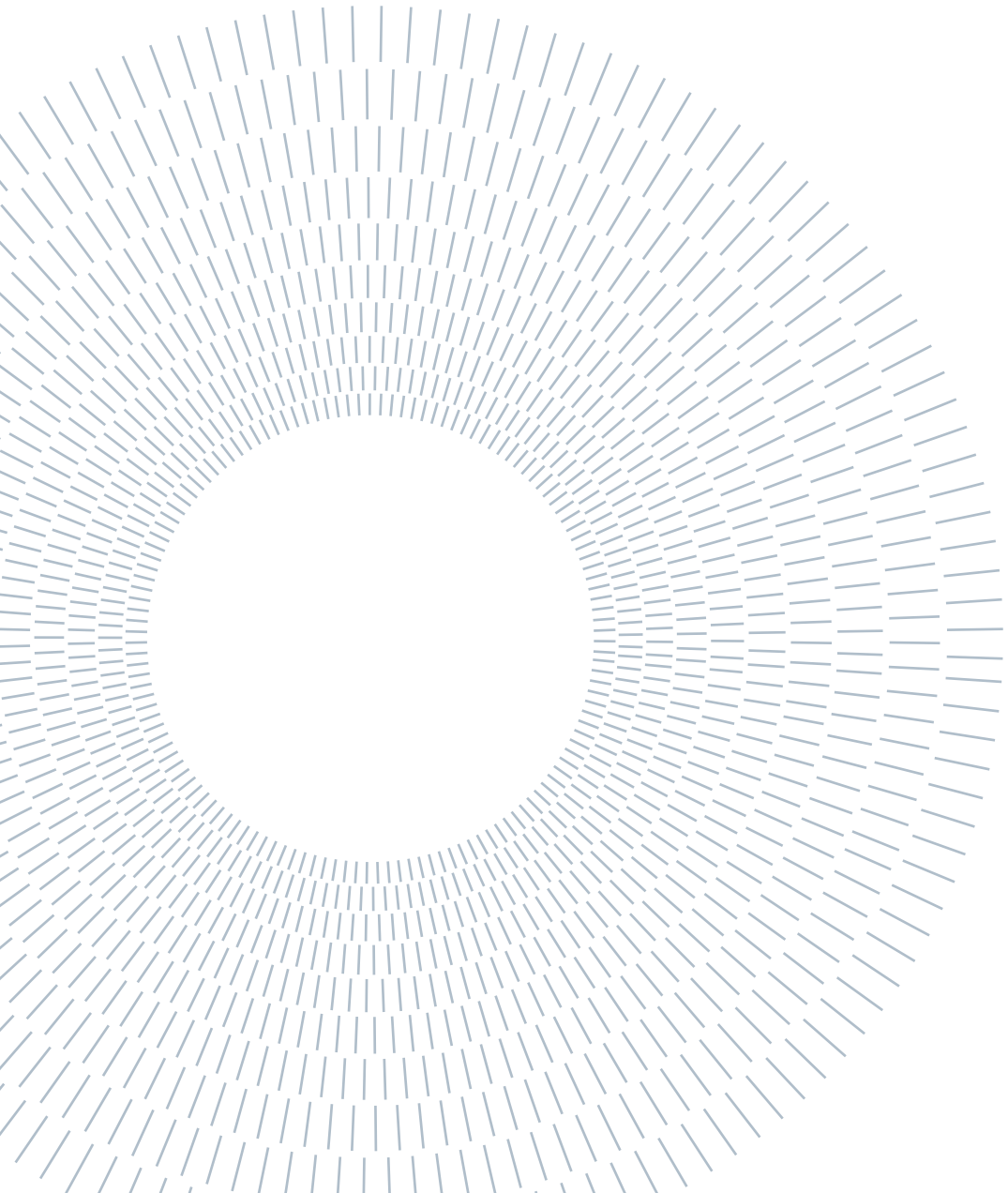
[17] Wikipedia. Boilerplate code, 2022. URL https://en.wikipedia.org/wiki/Boilerplate_code.

[18] N. Yosef. The ugly side of redux, 2017. URL https://codeburst.io/the-ugly-side-of-redux-6591fde68200.

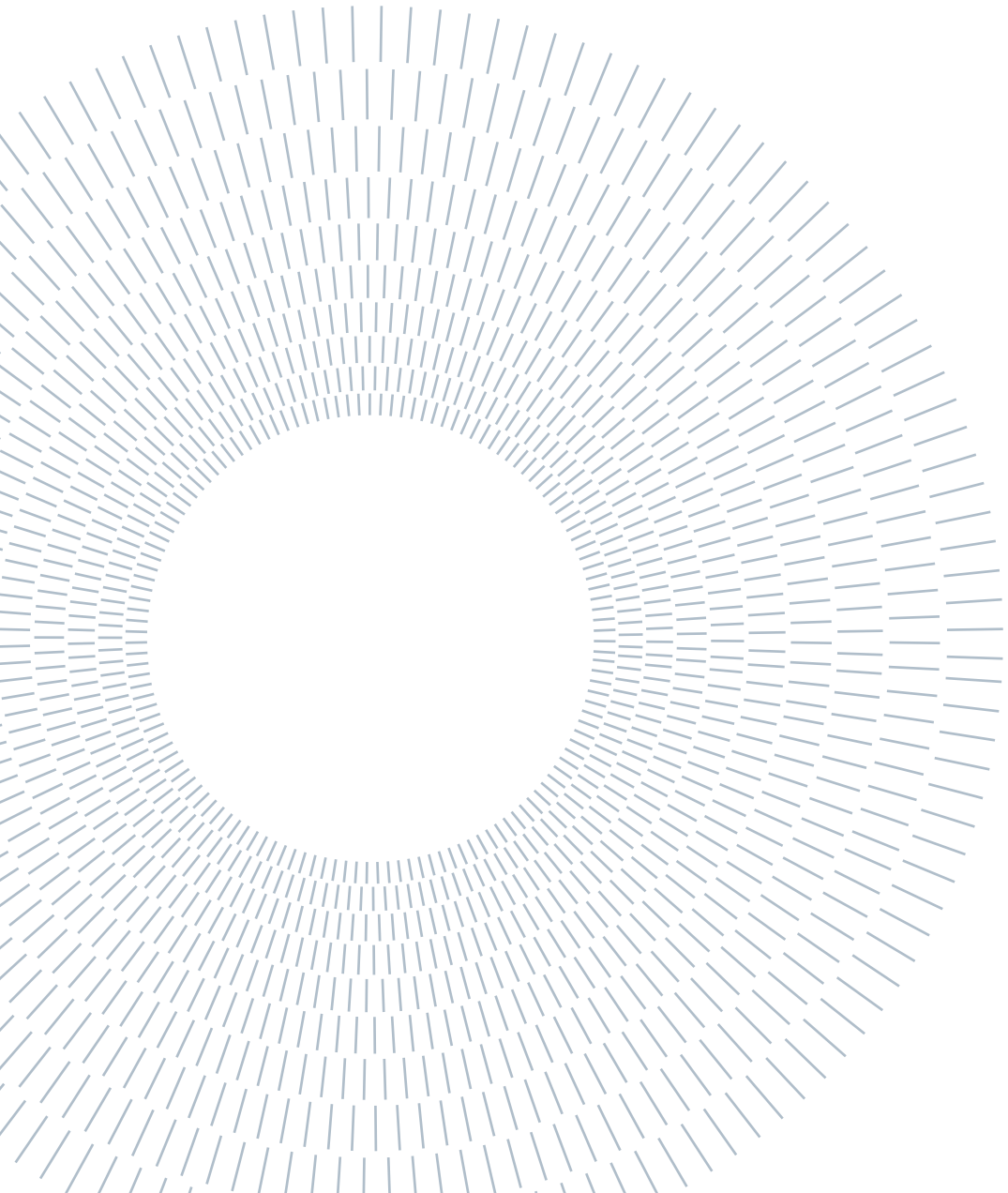
List of Figures



List of Tables



List of source codes



Acknowledgements

Ci tengo a ringraziare Lara, la mia compagna, che mi ha supportato per tutta la durata della tesi e del percorso al Politecnico. Mi è stata vicino nelle fatiche e mi ha sostenuto durante le delusioni. Ha sempre trovato le energie per aiutarmi a capire dove migliorare anche se a volte difficile.

Ringrazio i miei genitori che non hanno mai smesso di credere in me e di spronarmi a dare il meglio; che hanno saputo essere lungimiranti e trasmettermi il valore di questo percorso, anche quando non ne comprendevo a fondo l'importanza.

Ringrazio i miei amici che non hanno mai smesso di volermi bene e di cercarmi nonostante questo lavoro mi stesse portando via tempo e energie, prima dedicate a loro.

Ringrazio Paola Cigno che mi ha assistito con la lingua inglese, con impegno e interesse, durante tutta la stesura della tesi e che ha fatto sì che questo ultimo percorso fosse ancor più istruttivo e avvalorante.

Ringrazio il Politecnico e tutte le persone che ho incontrato durante questi anni che mi hanno insegnato come risolvere molti problemi, di natura ingegneristica e non.